UiO : Department of Informatics
University of Oslo

Requirements and Analysis of Extended HTTP Digest Access Authentication

Henning Klevjer, master thesis spring 2013

Abstract

Authenticating to an online service is usually done by providing a username and password in some protected form so that the server can verify that those credentials correspond to a registered identity authorised for access. For the average Internet user, managing one's online identities is challenging. Nearly every password protected web service advises its users to come up with a sufficiently complex password, which is not to be used elsewhere. As the number of associations with online services increases, so too does the number of online identities that the user has to remember. *Identity overload* is one of the greatest challenges for Internet users. The result may be that the user reuses difficult passwords for those of his online accounts that protect high value information, and uses passwords that are easily remembered (and easily guessed) for the protection of lower value information.

The main goal of this thesis is to investigate local user-centric identity management and propose a simple, secure and user friendly authentication mechanism. The mechanism relies on an external offline personal authentication device called "OffPAD", which acts as a trusted platform external to the terminal. From this device, the user may authenticate to services and manage his online identities. We argue that the approach of handling critical actions on an external secure device provides increased security and usability with regard to both the authentication process itself, as well as the storage and handling of identities.

The OffPAD device can be used to automatically authenticate its holder to any supported web service to which he or she is registered. We will present an extension of the HTTP Digest Access Authentication scheme that facilitates unobtrusive and automated authentication, while still adhering to password policies. We will look at how we can increase security and suggest improvements for modernizing the (ageing) digest authentication standard in particular, with regard to storage and handling of credentials. We will also discuss how identity management can be more user-centric, thus user friendly, alleviating the cognitive load of managing passwords.

HTTP Digest Access Authentication is used as the authentication scheme in every example and in the prototype implementation. It was selected for its simplicity, extendibility and abilities: especially its ability to function with both clear text and hashed user credentials at the endpoints.

Acknowledgements

I would like to express my gratitude to Prof. Audun Jøsang at the University of Oslo. He has taught me most of what I know of information security, and has helped me a great deal.

I would also like to thank Kent Varmedal, Magnus Evensberget, Ali Ahmad and Kristoffer Jensen, who survived my presence, and the working conditions in the infrequently ventilated fourth floor computer lab.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Local user-centric identity management can alleviate the pains related to *identity exhaustion* at the individual level [18]. The number of identities an *internet surfer* accumulates on the web quickly exceeds the number of identities she can keep track of on her own. This is a classic problem of bad usability, often leading the user to circumvent best practice: writing down passwords or reusing the same password for many services [2].

A sufficiently disconnected and secured external device can be used to make management of identities more efficient, secure, and scalable for each user. Such a device can work as both a local user-centric identity management system and an automatic authenticator. Bundled with implementations of the appropriate authentication algorithms, it is able to manage every one of the user's online identities, and at the same time automatically authenticate the user to his associated services. This device is called an OffPAD, a *mostly offline* Personal Authentication Device, with capabilities such as automatic authentication and secure identity management. Letting the device handle this work relieves the user of the mental strain of password management and will in most cases completely remove the time penalty associated with user interaction during authentication. Also, as authentication is done from an external device and the procedure passes only protected information through the client computer, no password or critical data is exposed in clear text on the computer screen or in the memory of the client. This, of course, requires authentication schemes that do not transfer plain text passwords. In an ideal situation, the holder of the described OffPAD device will never need to manage any of her credentials, or at any point know more than a single password, namely that which unlocks the device.

1.2 Research questions

The problems addressed in this thesis are related to identity management, access control and security usability. The main problem is that of password fatigue, described in [18, 15]. Another issue with contemporary password

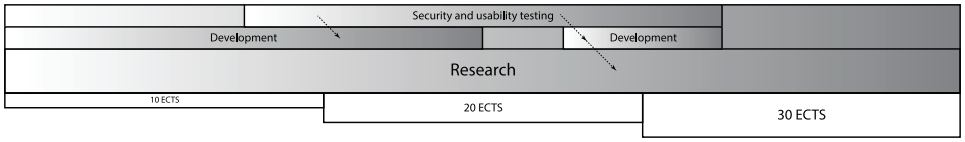


Figure 1.1: High-level research progress

schemes is that they require the user to enter his password on his computer through the keyboard, exposing the password in local memory (see figure 2.4). Theoretically, malware can examine memory segments of a web browser and extract the password entered. A more widespread threat is that of *keylogging*, where each keystroke is logged as the user is typing. This concern is expressed and discussed in detail in [21].

We will also investigate whether *original passwords*[1] are still needed when identity management happens at a device rather than in the brain of the user. In chapter 5 we elaborate on password protection and argue that with our proposed authentication scheme, passwords (as they are used in the original scheme) are obsolete.

Below we list a selection of important questions that will be assessed during the thesis and ultimately concluded in chapter 6:

1. What opportunities does the OffPAD offer?
2. What has already been done in the field of local user-centric identity management?
3. What security requirements must be considered for a contemporary user authentication scheme?
4. What is the state of HTTP Digest Access Authentication?

1.3 Process and scientific method

The work carried out in this thesis project has been done loosely following an iterative and incremental development practice, both for the development of the software and writing the report. The software product has been followed up by the LUCIDMAN project through its monthly team meetings. A sketch of the software was defined early, but many adjustments were made throughout the period. In the early stages of the process, planning and requirement analysis was done while investigating the limits and possibilities of the provided hardware (TazCard[2]). A working prototype, able to authenticate a user to an HTTP Digest Authentication protected server was presented on March 15, 2012, at a LUCIDMAN project meeting in Caen, France.

In the third quarter of 2012, it was decided to discontinue the development of the TazCard application and work toward deploying the

[1] I.e. passwords that are made up by the user alone, and that he has to remember.
[2] The TazCard is a Java enabled handheld device produced by French TazTag.

authentication system on another device, the TazPhone. However, because of delays in production of the TazPhone device, no working prototype was finished within the time frame of the thesis work.

1.4 Scope

This thesis' main topics are secure identity management, user authentication and authentication mechanisms. In particular, we focus on the implementation of an automatic authentication service for HTTP Digest Access Authentication and discuss its qualities. The software developed as part of the thesis work (described in chapter 4) is limited to work with the HTTP Digest Access Authentication specified in RFC 2617 [9] and the security extension of this algorithm, which we propose in chapter 5. Theoretically, the OffPAD device will be able to handle authentication by all schemes that rely on client-server communication.

1.4.1 Assumptions and restrictions

We restrict our view of HTTP Digest Access Authentication in that we do not support backwards compatibility with earlier (more vulnerable) versions of the algorithm[3]. The `MD5-sess` authentication mechanism and integrity protected authentication (`auth-int`) are also not considered, for brevity. We also assume that server authentication (authentication by the user of the server) is not a concern – this thesis focuses on user authentication only. We thus assume and rely on that the identity of the remote server is correct. Concerns of server authentication are addressed within the LUCIDMAN project by Kent Varmedal in [41]. Further, as a consequence of our client-side focus, we do not use or consider the `AuthenticationInfo` header (or its contained information) that is passed to the client upon successful authentication; neither in this report or the software solutions. This header is originally used for notifying the user agent (the web browser) that authentication was successful.

1.5 Related work

There are a number of devices developed in the field of identity management, each covering a portion (but not all) of the proposed OffPAD. In [23] Singer and Laurie take position on how the identity management problem should be handled, and present requirements for a device to alleviate identity overload, similar to the OffPAD device. Below we present some of the identity management and authenticator devices that are similar to the OffPAD. In section 2.2.4 we present the OffPAD and in section 3.2 we describe how the OffPAD differs from the other, similar devices.

[3] RFC2617 offers backwards compatibility with the less protected RFC2069, and a compromised server may advertise support for other less secure schemes.

1.5.1 The Pico

In [35], Frank Stajano introduced the Pico, a small device facilitating user authentication to any supporting service, both online and offline, both in a network context and "real life". This makes the Pico equally applicable in authenticating its user to a burglar alarm, Facebook or a point of sale.

> Imagine we could afford to start again from scratch, without concerns for backwards compatibility, and that our primary directive were to invent a user authentication method that no longer relied on users having to remember unguessable secrets. [... The Pico] is presented as a clean-slate solution, without regard for backwards compatibility. [35]

The OffPAD does not attempt to range as far as the Pico with regard to heterogeneity of services. In turn, the OffPAD scheme benefits from that it does not demand any radical server-side changes to function. The differences between the OffPAD and Pico designs are thoroughly discussed in section 3.2.1.

1.5.2 Nebuchadnezzar

Nebuchadnezzar, or *the Neb*, is a 2008 idea by Singer and Laurie. In [23] they argue that attempting to establish a trusted path of communication between a "general purpose" operating system and a server is a bad idea. It is not possible to assume a that a client running a general purpose operating system is sufficiently secured. They also present another unreasonable extreme: trying to do all of one's everyday computing on a minimal, secured, locked down operating system. They argue that the only sensible solution to secure computing is a combination of the two: a compromise between highly secure and highly usable. The position paper further describes the schematics behind a trusted device, the Nebuchadnezzar, which much like the OffPAD is an external device, maximally reduced with regard to features. The OffPAD may be seen as a physical implementation of the Nebuchadnezzar, with an even narrower (or more specialized) set of functions. In section 3.2.2 we elaborate on the differences between the OffPAD and the Neb.

1.5.3 MP-Auth

In 2010, Mannan and Oorschot [26] presented the MP-Auth (or Mobile Password Authentication) protocol. A solution for moving password authentication (and not the passwords themselves) to a personal device, protecting them against being collected by malware. In this protocol, a direct SSL tunnel is established between the user's mobile phone and the server to which he will authenticate. The user's password or credential is then entered on the phone and transmitted, protected by the SSL tunnel, to the server, authenticating the user [26]. The MP-Auth solution relays the communication and entering of a password to a mobile phone, but does not provide the benefit of identity management.

1.5.4 The commercial state of the art

Some devices are complete identity managers, letting the user manage his credentials on the device, but without the ability to authenticate directly to his services. Some are authenticator tokens, using *possession of the device* as an authenticating factor[4]. Below, we discuss the current commercial state of the art in user-centric identity management. Table 1.1 shows a comparison. At the time of writing, there are no commercial devices sufficiently matching the description of an OffPAD, and no physical implementations of the *Nebuchadnezzar* described in [23].

Name	Created / Issued by	Category	R
bUnit	Bloomberg L.P.	A	✗
Mandylion token	Mandylion Research Labs, LLC	IdM	✓
YubiKey	yubico	A(MF)	✓
Browser / SW password manager	Various	IdM+A	✓
Online password managers	Various	IdM+A	✓
BankID	FNO (Finance Norway)	A(MF)	✗
SecurID	RSA	A(MF)	✗

A = Authenticator | IdM = Identity Manager | MF = Part of multi factor mechanism | R = Reconfigurable

Table 1.1: Current commercial authentication and identity management devices

bUnit

The bUnit[5] is an authentication token made specifically for customers of Bloomberg Professional. Using its embedded fingerprint scanner, the user can authenticate to Bloomberg's Professional service from any computer, thus providing their customers with mobility between terminals.

Mandylion Token

Mandylion Research Labs developed their Token following standards and regulations by NIST, NSA and the US Military [25]. The Token can be used as an identity manager for up to 50 usernames (or account names) and password pairs. It should be noted that the Token has an upper limit of 14 characters per password. The Token is not an authenticator device, but is reconfigurable in that its contained passwords can be modified. The functionality of the Token is limited to presenting an authenticated owner with the contained password. It does not do authentication, but relieves password fatigue and advocates against reuse of passwords.

YubiKey

YubiKey is a popular device for user authentication. It is mainly an OTP code generator, sending transient passwords to YubiKey supported web

[4] I.e. "something you have".

[5] http://bloomberg.com/bunit/

sites. The YubiKey has several other applications, such as storing a static password, challenge-response, etc. As the YubiKey is only capable of storing a single password at a time, we do not consider it an identity management device.

Browser password managers

All contemporary browsers have some sort of password management service. These are recognizable by their inquiring the user to "remember this password". They are able to store passwords for most of the common username and password schemes, such as HTTP authentication. Some automatically fill in HTML forms. While most are lacking in their quality of protection, many allow a master password to be set, thus allowing encryption of the password storage. Thus, any Firefox identity store not protected by a master password can be collected by a Trojan or other malware, and the passwords can be decrypted at another location. However, the quality of protection provided by a master password is as usual dependent on the quality of that master password. Mounting brute force or other guessing attacks is trivial.

Software password managers

Software password managers such as LastPass[6] work in the same way as browser password managers in that they store passwords either on the user's computer or at their own online location. While still only requiring a master password to unlock every password, these services usually provide additional protection. LastPass generates a master encryption key for protection of the managed passwords. This master key is generated with the PBKDF2 key derivation function using SHA-256. Key derivation functions are discussed in section 2.3.1. It is worth noting that while the PBKDF2 specification in RFC 2898 suggests 1000 iterations of the KDF one way function as an *absolute minimum*, LastPass only uses 500 iterations [20, 22]. Also, as managed passwords are released to the computer upon use, they may be read from memory in clear text (see figure 2.4 and [21] for details).

BankID

BankID is an effort to make online banking in Norway easier and more secure. It is developed by Nets and is through the organization *Finansnæringens Fellesorganisasjon* (FNO), owned by 180 Norwegian banks. A BankID member bank may issue their customer a BankID OTP device, that he can use as an additional authentication factor when authenticating himself to the issuer bank, and to other member banks with which he has a customer relationship. The BankID OTPs are also used for other actions originally requiring written consent, such as entering a loan agreement, applying for credit cards, and the like. The digital signatures provided

[6] https://lastpass.com/

to such as transactions or agreements apply non-repudiation and message authentication to these.

SecurID

SecurID comprises several products which are all OTP authentication tokens. A SecurID token authenticates the holder to one service, to which the token is configured. Some SecurID tokens have other functionality as well, such as challenge-response authentication and email signing.

1.6 Organization of the report

Chapter 2 explains technical details needed for understanding the software systems and the later discussions. The most important theories discussed in this report are identity management (section 2.2) and HTTP Digest Access Authentication (section 2.5). A thorough presentation of cryptographic hash functions and key derivation functions (KDF) is given in section 2.3.

In chapter 3 we discuss the problems presented in section 1.2, and give various solutions, weighing them against each other. We conclude by arguing on why the chosen solutions are the preferred ones and what their limitations are, if any.

Given the technical background from chapter 2 and the overview of the project from chapter 3, we use chapter 4 to exhibit the software implementations and discuss them in detail.

To ensure the quality of the software implementation with regard to security, the software has been scrutinized through several prototype stages. Details of the security tests are presented in chapter 5, along with suggestions on how to secure operation and the consequential steps taken. Chapter 5 is not intended as a total overview and solution to all of the scheme's weaknesses.

Chapter 6 concludes the thesis and raises some open issues and ideas that may fit for future research.

1.7 The LUCIDMAN project

The LUCIDMAN project[7] is a 2011-2013 Franco-Norwegian research project on security usability. Below is a short presentation of the collaborators and their expertize/contributions to the LUCIDMAN project:

> The objective [of LUCIDMAN] is to investigate the security and the usability aspects for client side management of user and service provider identities. The project will investigate an identity management model which can provide enhancements to security challenges associated with poor usability and scalability of managing identities on the client side [38].

[7] LUCIDMAN is an abbreviation of **L**ocal **u**ser-**c**entric **Id**entity **Man**agement.

Vallvi AS

Vallvi is a Norwegian company that focuses on consulting and business development in the ICT sector. In the LUCIDMAN project, Vallvi contributes with management and business development.

ENSICAEN GREYC

GREYC is a research laboratory at École Nationale Supérieure d'Ingénieurs de Caen (ENSICAEN). With over 220 members GREYC represent knowledge in a wide variety of fields such as computer security; identification (biometrics etc.); microelectronics; algorithms. Their research group *E-payment & Biometrics* is a contributor to the project with knowledge on authentication, privacy and electronic payments. French CEV is a developer of customer loyalty solutions using smart cards, magnetic cards, contactless devices, etc. Contributions of CEV include integration of an *identity selector*, into the identity management system.

TazTag

TazTag is a French developer of secure contactless devices such as the TPH-ONE Android mobile phone, the TazCard handheld device, and the TazPad tablet. These devices are enabled for confidential communication and a god fit for the security requirements of the applications developed in the project. Thus, TazTag contribute with devices, API and documentation for application and prototype development.

Tellu

Tellu is a Norwegian software company working on mobile solutions. They provide UI design, prototyping and testing of the project's applications.

Department of Informatics, University of Oslo

The Department of Informatics at UiO has a wide field of research with a growing interest in information security. Lead by Dr. Audun Jøsang, the UiO group of the LUCIDMAN project will contribute with two Master's theses on security and usability: "Cognitive entity authentication with Petname systems" by Kent Varmedal [41], and the thesis of this report.

Chapter 2

Background material

This chapter is intended as an introduction to the most important theories, covering technologies and methods mentioned and exemplified in this report.

2.1 Terminology

2.1.1 Authentication

Entity authentication is the act of an entity[1], in which it proves its identity to a system (the *verifier*) by providing it with sufficient information for that system to substantiate its veracity. An entity is usually assigned certain privileges or authorizations, for which it must be authenticated to exercise by.

For the entity to be able to authenticate, it must first be registered with the verifier. Ordinarily for web services, a set of credentials is supplied by the user when he first registers to the service – so-called *leap-of-faith* authentication [3], or *TOFU*, Trust On First Use. Registration under the *TOFU principle* is a critical action in that the credentials supplied in the registration phase provide the highest level of assurance that the system can ever have in the user's authenticity. In this case, the verifier knows nothing else than that the user, upon authenticating, is the same one that once registered. It must trust that the credentials supplied reflect the true identity of the user. On the other far end, banks may require that their customers register by providing a physical proof of identity, such as a passport or other national means of identity, even meeting up in person. Here, the identity is already established in a national registry and provides a higher assurance of correct identity upon registration. Following, each authentication will provide, at most, the assurance with which the user registered.

[1] An entity is either a person, organization or object (such as a file, directory or data unit)

2.1.2 TOCTOU

Time Of Check / Time Of Use (TOCTOU) in authentication describes the distinction between two events in access control. The *time of check* is the instant in which the identity of some entity is verified by the system. The *time of use* is the instant in which the access control system looks up the authenticated identity in the access policy and approves access if the identity is authorized. If the two events are separated in time, the Authentication Assurance Level (AAL) is lower at the *time of use* than if they are sequentially and atomically executed. The time between check and use can be called $\Delta TOCTOU$.

As an example, consider the verification of bank transactions, or the requirement to re-authenticate upon password changes in most systems. These are situations where the TOCTOU principle holds. The access control system's assurance of the authenticated identity has dropped below the level demanded and the system must be reassured in order to execute the critical action. In scenarios where a user is *logged in* to a system, that system can only be certain of the user's identity at the time of his authentication (*time of check*), as an adversary may overtake the session, e.g. if the authenticated user has left the system without logging out of his session.

2.1.3 Continuous authentication

Through continuous authentication, the system can be certain of the entity's identity through an entire *authenticated session* or every time authentication is required. It enables the authentication system to be stateless, i.e. a user is not either *logged in*, or *logged out*. Rather, the system is in a state of constant doubt – the AAL is low. The system can be *reassured* of the entity's authenticity either steadily (e.g. through biometrics, such as in [29][2]) or at regular intervals (e.g. re-authenticating after a time out or when needed (at the *time of use*)). The apparent benefit of continuous authentication is that since being confirmed throughout the session, the system's AALa is higher over time.

2.1.4 Authorization and access control

Authorization is the act of defining an *access policy* for an entity's access to an object[3]. Access policies define what are the entity's capabilities in the system, i.e. what protected objects or realms of protected objects that the entity is entitled access to. Authorization may be specified using the common file system permissions (read, write, execute, append) or more intricate permission systems, such as the Bell-La Padula model.

Access control is the enforcing of an access policy on a system. The access policy usually defines either a *discretionary* or *mandatory* access

[2]The author presents this technology in http://youtu.be/pqtZyTyBB7k.

[3]Here, *entity* may be interpreted as several entities, such as a group of entities sharing a role (role-based access control) or a common set of rules (rule-based access control).

control. In a system enforcing discretionary access control (DAC), the access control is done at the *discretion* of the owner of the data. This way, if a subject (e.g. a user or system) creates an object (e.g. a text document) on a system, he becomes its owner and delegates access (i.e. authorizes access) to that object for other subjects as he sees fit. Authorizations are delegated to *identities* or *groups*, making DAC identity based access control.

In a mandatory access control (MAC) policy, there is one access policy for the entire system and authorizations are delegated based on an object's security label. This makes mandatory access control particularly applicable to military structured systems, where objects may be labelled "confidential", "top secret", etc. [10], and discretionary access control applicable to systems with a flat hierarchy.

The phases of access control and the relationship between the terms *authorization*, *access control*, *access approval* and *authentication* is emphasized in figure 2.1.

- Registration phase
  - Registration – Identifying an entity based on provided attributes.
  - Provisioning – Creating a (system-) unique identity for the entity and collecting the corresponding authentication factor.
  - Authorization – Defining and persisting an access policy entry, describing the user's access rights.
- Operation phase
  - Identification – Collecting the identifiers of an entity, mapping it to a digital identity.
  - Authentication – Having the entity prove that it is the owner of the assumed identity.
  - Access approval – Acting on the authenticated user's authorization ("opening the gate").
- Termination phase
  - De-registration – Removal of the digital identity from the access control system.
  - Revocation – Modifying the access policy to no longer contain entries on the de-registered user.

Adapted from [16].

Figure 2.1: The phases of access control

	SP perspective	User perspective
User ID	1A	1B
Server ID	2A	2B

[16]

Table 2.1: The four processes of identity management

2.2 Identity management

Identity management covers all aspects of the creation, use, storage and revocation of digital identities. A *digital identity* describes, and maps uniquely to, an *entity*. The digital identity is usually referenced by a username, or other *attributes* that alone, or in assembly are unique, distinguishing the entity from others. In biometric authentication, the attributes of the entity are attributes of the actual owner, i.e. finger print. Figure 2.2 describes some attributes that can be used to uniquely identify a business (name and registration number) and a person (name, height, iris pattern, voice and fingerprint). Note that in biometric user authentication, the biometric attributes of the user may be used for both identification and authentication, i.e. the biometric data acts both as identifier and authentication factor.

In [16], Jøsang presents four different processes of identity management: two from the perspective of the service provider (category A, or *server-centric identity management*) and two from the perspective of the user (category B, or *user-centric identity management*). Both the service provider and the user have a notion of managing user identities and server identities at their location (see table 2.1).

From the service provider's point of view, managing users' identities (type 1A) is usually done by a customer relationship management system (CRM), or in smaller systems simpler access control systems. These identities are often system unique, complicating the users' management of their own online identities (type 1B) which normally spreads across an infeasible number of services. Type 1B is an important topic of this thesis, historically a much overlooked topic.

Type 2A describes the service provider's management of its own identity(-ies). On the Internet, a server's identity is its URL, sometimes coupled with a public key (in an X.509 certificate) for encrypted access, and server authentication through hierarchical validation of that public key. Each user's handling of remote services' identities usually happens through the browser, or by the user as proposed in [37], following the Petname systems.

2.2.1 The silo model

The simplest and most straightforward model of identity management is the silo model. Credentials of all users are centrally stored at the identity provider's servers, and at login time, the credentials that the user supplies are validated against local records. Using the silo model, every user must have a separate identity at each identity provider's "silo".

Figure 2.2: Attributes (identifiers) of two entities

2.2.2 The federated model

Proving cumbersome to the user, some identity providers improve on usability by merging their identity silos into an *identity federation*, where identities are shared between many identity providers, providing cross-domain validity. This is done in agreement between the affected identity providers. In most cases the identity is first registered at the federated identity provider and authentication happens on the service provider's location using the federated identity provider's authentication API.

The federated model is a means to alleviate the obvious usability problem of the silo model: the number of identities. Where the silo model is autonomous in its realm, the federated model strives generality. Federated identity providers aim to create a digital identity that is portable over multiple heterogeneous services: that the owner is eligible for authenticating to different service providers by the same federated credentials. Authorization to federated identity management systems is done based on a shared set of access policies.

Single Sign-On

Single Sign-On (SSO) differs from the federated model in that it is the authentication process that is portable, rather than only the identity. When authenticated by Single Sign-On, a user is automatically approved access to all the protected realms he enters, based on the shared identity's authorizations. The user usually authorizes the services to utilize the SSO capability of the identity provider to which he is registered. When such a service employs an external federated identity provider, authentication happens at that identity provider's location. As an example, Google Account provides SSO capabilities for all Google's internal services, such as GMail, Google Play, etc. and the user's Google identity can be federated

across e.g. the Stack Exchange network. The federated Google identity may then used by the Stack Overflow web server to authenticate the user over the entire Stack Exchange Network (of which Stack Overflow is a part). That is, the user is *SSO* signed on to the Stack Exchange network with his federated Google identity.

Federation of heterogeneous services and companies in particular has never really taken off, probably due to the trust issues that arise when an identity is shared between service providers.

One can argue that identity providers such as Google, Facebook (Connect) and OpenID have had a huge impact on identity federation for large niche applications such as comment fields on online newspapers and message boards. However, these services are rather similar and not reaching across heterogeneous domains or domains requiring a high level of authentication assurance. Using Facebook identities on a hospital computer to access patients' journals will definitely be considered an inadequate solution, now and in the future.

2.2.3 Local user-centric identity management

Normal user-centric identity management requires that the user remembers each identity on his own. Consequentially, writing down or reusing secret credentials are common problems [1]. As there is no reason to believe that there will ever be one unique *online identity* per Internet user, one could try to solve the problem locally.

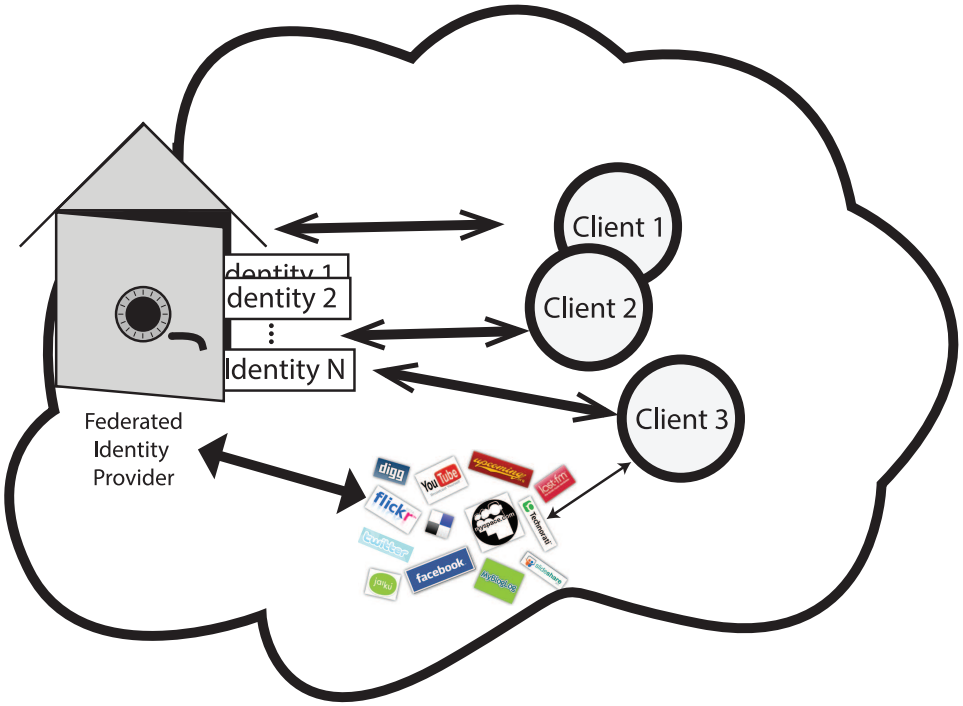


Figure 2.3: Identity federation

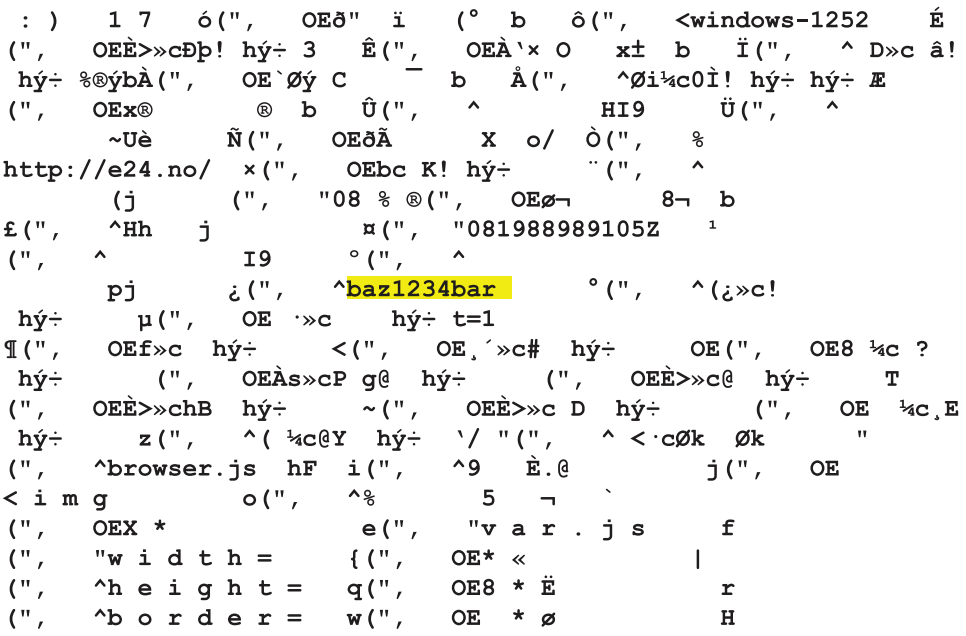


Figure 2.4: A password exposed in memory

Storing identities on a secured device is a possible solution. In [17] Jøsang and Pope describe the Personal Authentication Device (PAD), a secure device that is "mostly decoupled" from the computer. The PAD is used as an identity management system [4] to which the user authenticates once (with a PIN number, passphrase or similar). For one *session*[5] on the PAD, the user can authenticate to every supported remote service automatically. This is done by securely transmitting the credentials or proof of credentials between the PAD and the server.

2.2.4 OffPAD

In [15], Jøsang describes a more secure PAD, the OffPAD. The OffPAD is a personal authentication device (PAD) that is sufficiently disconnected (*Off*line). By reducing the device's connectivity, a decreased number of attack vectors follow. This decoupling from networks improves security on the device, as it is less vulnerable to outside attacks.

Critical for an OffPAD is that the transfer of information from the PAD to the ultimate destination is done securely. An improvement over authentication using the computer directly is that in the latter case the password is written through the keyboard and exposed in memory, while with the OffPAD, credentials are protected with a one-way function before moving out of the device. In figure 2.4 we show a password as exposed in a web browser's memory.

As a side note, consider the amount of attack vectors in a mobile

[4] type 1B and 2B in table 2.1

[5] Or some limited amount of time.

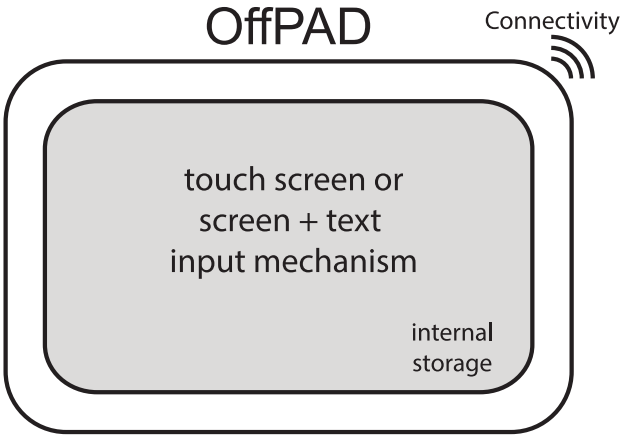


Figure 2.5: A possible OffPAD design

phone, which is (and strives to be) extremely feature-rich. Consequently (in addition to its widespread use, of course), the mobile phone is becoming a vulnerable platform [19] and thus cannot be used securely in conjunction with our software. Below are some suggestions for an OffPAD presented in [21]. We will describe mobile phone implementations of the OffPAD further in section 2.2.5.

1. Limited connectivity - we suggest NFC or other physically activated communication (contactless).

2. Secure element - an infrastructure for secure messaging and storage such as described in ISO 7816-4[6].

3. Access control by PIN number or password protection, restricting unauthorized access to the device. For additional device protection (two-factor authentication) a combination of biometrics and an access code can be used.

2.2.5 Using a Mobile Phone as the OffPAD

The current trend of mobile phone malware strongly indicates that the mobile phone is joining the computer as a vulnerable platform. *"In 2011, the Juniper MTC identified a 155 percent increase in mobile malware across all platforms, as compared to the previous year"* [19]. The number of features in mobile phones, especially connectivity features, increase the number of attack vectors, thus the overall attack surface of the device. In light of this trend, which we assume will not turn to the better, we deem the OffPAD application unfit for implementation of a mobile phone operating system. As Singer and Laurie argue in [23], there is no way to establish a trusted path between a general purpose operating system (which we from

[6] http://www.iso.org/iso/iso_catalogue_catalogue_tc/catalogue_detail.htm?csnumber=36134

the above trend assume that the mobile phone is, or is becoming) and a server.

As a counterexample, the French company TazTag which specializes in secure contactless devices are developing a mobile phone (TPH-ONE)[7] which is said to be able to separate the secure element from the phone's operating system (Android), in having a *secure state*, that can be toggled on or off physically by the user when required. The *secure state* is a security context in which the phone works with the secure element only. The secure element is capable of handling encryption, and hashing of the credentials used for authentication. In this scenario, the phone is an OffPAD whenever it is in the secure state.

2.3 Cryptographic hash functions

A cryptographic hash function is a one-way function in which a value (bit string) of arbitrary length is converted to another value from a smaller set of values (commonly between 160 and 512 bit). The output of a hash function is usually called a digest or a hash (code/value), and is used to identify and differentiate between different data. Hash values can be presented as $H(x) = y$, where H is the hash function, x is the value being hashed and y is the resulting hash. Hash functions are used in data consistency; password based user authentication; distributed hash tables; message authentication; digital signatures; challenge-response protocols, etc.

A hash function must satisfy at least the following requirements [27]:

Ease of computation

Given some x, it is easy to compute $H(x)$.

Compression

The hash function H maps input values x of arbitrary length to output values $H(x)$ of the same fixed length.

Preimage resistance

For a specified hash code y, find its *preimage*. A preimage, x, is the value that is used to produce the hash code y or any other value that produces the same y, i.e. in $H(x) == H(x') == y, x \neq x'$, both x and x' are preimages of y. A preimage can be used to counterfeit message authentication codes (MAC's), or recover a password from a hashed password file. Preimage attacks violate the hash function's *one-way* characteristic. For a one-way function to be preimage resistant, it must be computationally hard[8] to find a preimage.

[7] http://taztag.com/index.php?option=com_content&view=article&id=104

[8] The term *hard* in cryptography refers to calculations that are infeasible to execute on today's most powerful computer systems and where a brute force attack usually is the most effective solution.

Figure 2.6: Counterfeit digital signatures

Additionally, a general purpose hash function must be resistant to the following attacks:

Second preimage

For a specified preimage x, find another preimage x', i.e. find a value such that $H(x) == H(x'), x \neq x'$. This attack may be used for creating counterfeit authenticated documents or X.509 certificates for web sites or software (applications, drivers, etc.). Resistance to such an attack is also called *weak collision resistance* and is the most difficult attack.

Collision

For a random preimage x, find another random x' such that $H(x) == H(x'), x \neq x'$. Collision attacks are only exploitable in situations where the attacker has some degree of control over, and flexibility in choosing input values.

Consider a scenario with a high number of documents that are semantically equivalent[9] and another set of such documents where the contents of each of the second set's documents carry a message with an entirely different meaning than in the first set. If a collision attack can be lead in such a way that the result is that one document's hash code is equal to that of another document which carries another (possibly adverse) message, the digital signature of either document is equal to the other, falsely verified documents can be produced. See figure 2.6 for an example in which collisions are used to create fraudulent X.509 PKI certificates for SSL/TLS communication. Resistance to this attack is also called *strong collision resistance*, as both preimages are randomly selected. Thus, it is more likely to succeed than a second preimage attack.

In [27, sec. 9.3.4], Menezes et al. argue that the ideal computational security level (the number of attempts needed for a successful attack) for a collision resistant hash function is $2^{L/2}$ and 2^L for both preimage and second preimage attacks. In the last two, the attacker is constrained by his requirement to find a specific preimage. This is not the case in collisions, where any preimage is sufficient.

2.3.1 Key Derivation Functions

Ordinary hash functions have many applications. Many problems are solved from the hash function's ability to quickly convert large amounts of data into a fixed size value, uniquely identifying the data. Version control systems, digital signatures and data comparison are among the applications that benefit from the speed of these highly efficient functions. When hashing a password for use in user authentication, for example using the MD5 hash function, the calculation itself is done in an incredibly short

[9] I.e. carries the same message, but each with different wording.

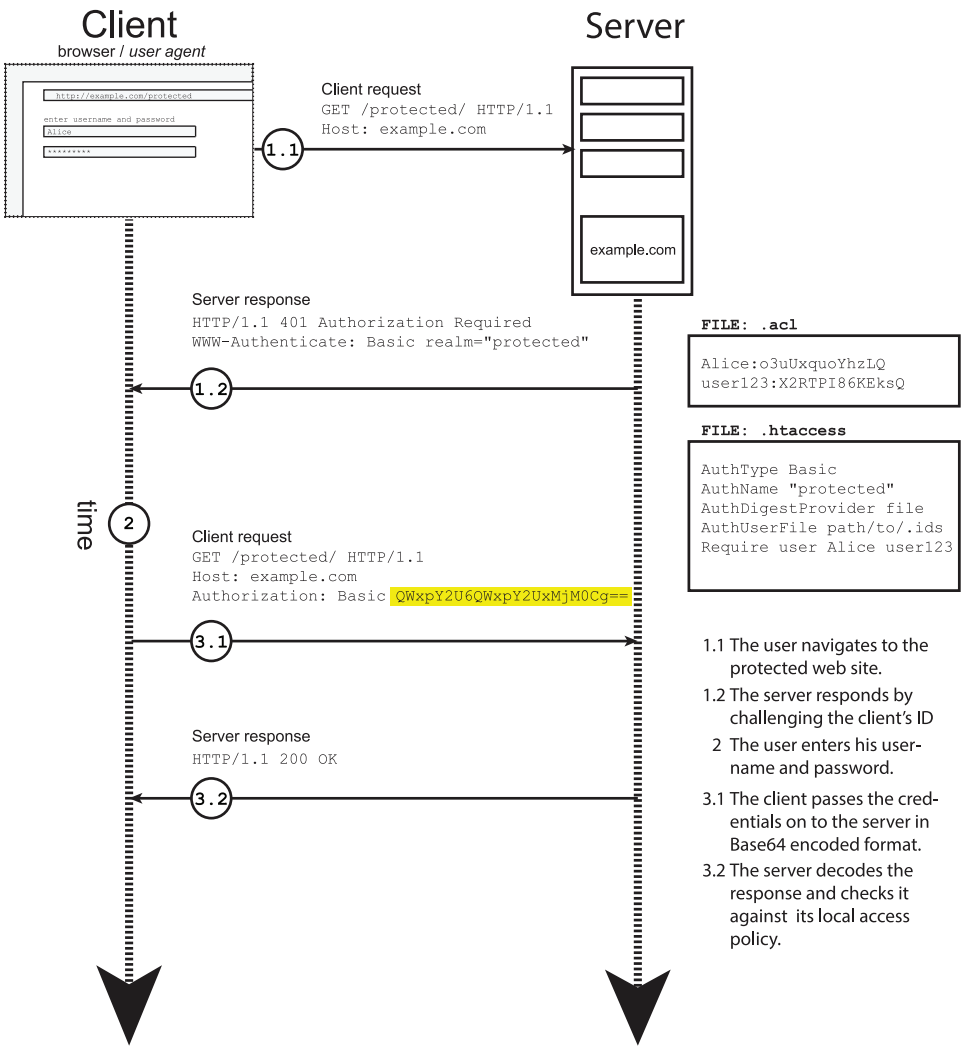


Figure 2.7: Basic Access Authentication

amount of time. If the intention is turned around, however, it is easy to see how fast a brute force attack may be carried out. In 2012, Jeremi Gosney reported brute force attacks using the MD5 function measuring up to 180 billion calculations per second. The attacks were carried out on five clustered servers, connecting 25 GPUs in total [12]. This means that the character space containing all 95 printable ASCII characters up to a length of eight characters (i.e. $\frac{95^9-1}{94} - 1$), or about 6.7 *quadrillion* character combinations, can be calculated[10,11] within a little over ten hours.

$$\frac{6704780954517120}{180000000000} = 37248,78 \text{ seconds } \approxeq 10,3 \text{ hours.} \tag{2.1}$$

Because of the effectiveness seen in brute force and dictionary attacks against hash functions, the need to slow them down was introduced [28]. Several thousand iterations of MD5 can slow down the calculation enough to mitigate most brute force attacks. The Password Based Key Derivation Functions (PBKDF1 and PBKDF2) [20] were introduced by RSA in 2000, but do not explicitly mention user authentication. bcrypt by Provos and Mazières (1999), and the more current scrypt by Colin Percival (2012) [30], however, specifically have user authentication in mind. Common to all key derivation functions is that they are heavier to calculate, thus provide a stronger protection to the password they protect. The PBKDF functions are used mainly to facilitate password based encryption by generating a key from a password, but every one-way key derivation function may also be used for user authentication. It is up to the identity provider to determine the workload of each hash calculation. Despite the long history of Key Derivation Functions, they have become popular only recently for storing passwords. By slowing the hash calculation down, each user's authentication may take tens, or even a few hundred milliseconds to perform, but this also applies for each attacker, who must use the same amount of time for each guessing attempt.

$$KDF(Key, Salt, Iteration) = DerivedKey \tag{2.2}$$

Servers protected by the original digest authentication scheme advertise their supported one-way function algorithms to the client in the `algorithm` field of the challenge. This enables servers to provide support of several others than the two algorithms specified in the standard (MD5 and MD5-sess). Advertising a KDF at the server side will benefit both the server and the client with additional protection of the password at both sides. If the algorithm used is implemented consistently (i.e. calculates the exact same values) at both endpoints, any one-way function or KDF should be transparently applicable to the authentication scheme.

Using KDFs not only protect the user credentials in transmit, they provide the same benefits to either communicating entity storing the

[10] Here we assume that there are no characters of length zero (we subtract $95^0 = 1$ from the original *geometric sum formula*). This leaves us with the expression $95^8 + 95^7 + ... + 95^1$.

[11] Amusingly, the set of ten letter passwords alone takes ten and a half years to calculate with the same speed. The calculation from equation 2.1 would take over four years on a desktop PC (assuming 50 million calculations per second).

credentials locally (i.e. the server and OffPAD). If a password database is breached, and the stored passwords are hashed by single MD5, and even salted, most passwords are recoverable in a reasonable amount of time. If the passwords are protected by a KDF and a reasonable workload is applied, brute force attacks will not be feasible. If a KDF uses 100 ms on a specific system to hash a dictionary password[12], it follows that the attacker requires over two hours on average to iterate a 150.000 word dictionary and locate the correct one, on the same system. Thus, KDFs provide better protection, even for poorly chosen passwords. Keep in mind that it may be reasonable to assume that the attacker is in possession of a much stronger system.

2.4 Basic Access Authentication

The Digest Access Authentication scheme, which is the inspiration of this thesis, the standard supported by the software implementation and the example in all use cases, is best explained by first elaborating its origin, HTTP Basic Access Authentication (short: Basic authentication). It is a scheme applicable to the authentication framework provided in HTTP and was introduced in 1996 as a part of the HTTP/1.0 protocol specification [4, chap. 11.1]. It was developed further and published as an Internet standard in RFC 2617 [9] in 1999. Basic authentication is a simple protocol for a server to authenticate a user requesting access to a protected area, called a *realm*. Basic Authentication was never regarded a secure scheme; already in 1996 ([4]) its authors cautioned users about its weakness: the secret user credentials are sent in clear text. To be specific, the username and password are Base64 encoded and sent to the server with no more protection. Base64 is an encoding scheme used for serialization of heterogeneous data items (text, binary strings, etc.) for transmission over the internet as text [13, p. 282]. The steps of Basic authentication are outlined below and can be followed in figure 2.7.

1.1 The user navigates to the protected web site.

1.2 The server responds by challenging the client's ID.

2 The user enters his username and password.

3.1 The client passes the user's credentials to the server in Base64 encoded format.

3.2 The server decodes the response and validates the credentials against its local credentials and access policy. Authenticated and authorized users are approved access.

[12] i.e. the password is in a dictionary.

2.5 Digest Access Authentication

HTTP Digest Access Authentication (short: digest authentication) is a 1997 web standard defined in RFC 2069 [8] and further in the obsoleting RFC 2617 [9] in 1999, for authenticating a client to a server. Its intended use is on the World Wide Web, but it is perfectly implementable on local resources, or in any situation where resource access over HTTP should be restricted[13]. Digest authentication was introduced as an extension to the former standard *Basic Access Authentication* described in [4, chap. 11.1], which is considered insecure [9, 13]. The most critical weakness of Basic Access Authentication is that passwords are passed in clear (Base64 encoded) over the internet. In practice digest authentication is used to enforce an access control policy on a certain realm and the resources it contains. Access to a realm protected by digest authentication requires each user to:

1. be registered with sufficient credentials (username and password) in the access control policy of the system enforcing the realm's access control (that is, *have authorization* for access to that realm). An access control policy is shown in figure 2.9.

2. be able to authenticate himself to the server using those registered credentials, i.e. provide them to the server upon request.

To understand digest authentication, consider this scenario:

A user wants to access some web resource at http://example.com/protected/. The `/protected/` directory (or realm) is protected with digest authentication, so that only authorized users can get access to it and its subdirectories.

Trying to access `http://example.com/protected/` initiates the following challenge-response communication between the client and server, over the HTTP protocol:

1. The client's web browser (*user agent*) issues a HTTP `GET` to retrieve http://example.com/protected/

2. Server responds with a `401 Authorization Required` HTTP status code, indicating to the user that he is currently not authenticated and that access to this resource is protected, requires access approval by the system[14] and that he is currently not authenticated. Along with the status code, the server passes a `WWW-authenticate` header, containing information needed for the system to calculate the correct response for the server.

3. The web browser interprets the `401` status code and prompts the user for the username and password, registered for the specified realm.

[13] The theoretical part of the scheme is applicable also outside HTTP.

[14] In the RFC specification, this stage alerts the need of what the `401` header confusingly refers to as *authorization*. What the system actually requires is that the user authenticates himself, so that the system can *validate his authorization* and either approve or reject access to the resource.

4. The entered credentials and the information extracted from the incoming `WWW-authenticate` header are hashed. The client issues another `HTTP GET`, now with an appended `Authorization` header, containing the response value (i.e. the previous hash) and other values.

5. The server receives the response value – a temporal proof of the user's identity and the name of the *protection realm* to which he claims to be authorized for access. As hashing algorithms are one-way functions, there is no way for an adversary to simply extract the passwords from the value. The protection of the response value relies on the quality (entropy) of the password, or at the very least on the preimage resistance of the hash function. At the server side, a credential hash that was stored locally in the registration phase and the challenge data are used to calculate another hash value by the same rules and algorithm as at the client side. If the server side calculation is equivalent to the one received from the client, the server can conclude that it is certain of the client's identity and validate it against the access policy. If the user is authenticated and authorized, the server responds with a `200 OK` status code and the contents of the protected resource that the user requested. If not, he is presented with another `401 Authorization Required` and given another attempt at authenticating.

In addition to cryptographic hash functions, several other security mechanisms were introduced with Digest Authentication. These are described in 2.5.1 and 2.5.2[15].

2.5.1 Replay attacks and the *nonce*

Basic authentication's big flaw is that the user credentials are minimally protected; when handled over an unencrypted medium they are readable by anyone who knows what to look for. After decoding the password, the attacker is in a position where he can get approved access by falsely authenticating himself to the remote server as well as at any other location the victim happens to be registered with the same password. Introducing hash functions to basic authentication converts the password in a non-reversible fashion – by the preimage resistance property of the hash function, one cannot extract the password from the hash[16]. The attacker can no longer directly deduce the password from the source, but he is able to reuse the values he captures.

Below is a scenario exploiting the first weakness of Basic authentication – called a *replay* attack, where credentials are collected by an attacker and sent to the server, authenticating the attacker as the credentials' owner.

[15]Some of digest authentication's security mechanisms have been omitted for brevity(cf. section 1.4).

[16]While the hash function itself cannot be reversed, weak hashed passwords can easily be recovered through an exhaustive search through the password's character space. We discuss this in length in section 2.3.1 and chapter 5.

Figure 2.8 can be used as reference. From the previous paragraph, we can see that protecting basic authentication by only substituting Base64 encoding with a one-way function, this form of authentication is also vulnerable to a replay attack.

1. Alice (the client, or victim) logs on to a remote server and authenticates using her username `Alice` and the password `Alice1234`. This produces the following response value in the `Authorization` header: `Base64("Alice:Alice1234") = QWxpY2U6QWxpY2UxMjM0Cg==`.
   The server decodes the header value and verifies the received password with the stored one. If the passwords are equal, she is authenticated successfully.

2. Harold (the hacker) is tapping into the network traffic, collects Alice's encoded `Authorization` header. He can either decode it and retrieve the password, trying it at many different locations, or replay the encoded header to the server directly.

3. Harold has falsely authenticated himself as Alice and gains unauthorized access to Alice's resources.

This is known as a replay attack. At the level of protection provided by Basic authentication, the `Authorization` header is presented to the server looking like it originated from the other authentic user. The server unwittingly identifies the login attempt as coming from Alice. Note that Alice has no way of noticing the attack: she is experiencing a *passive attack* from Harold, in which the communication channel is only read from. Alice's experience of the authentication process is not changed – HTTP authentication is a stateless authentication mechanism in which there is no notion of being *logged in*[17], which means that two consecutive authentications is not an unlikely event. The false authentication, however, may originate from another IP address than Alice's.

Countering replay attacks is done by introducing another value into the communication: the nonce. The nonce value is a *n*umber used only *once*. In HTTP Digest Access Authentication, this means for one attempt at authenticating. After a successful authentication, the same nonce is valid for a limited amount of time – a *nonce lifetime*[18]. The nonce is created at the server as a challenge to the user. If she is able to respond with that nonce (and it has not already been used) together with a correctly generated `response` value, she passes the challenge. This way, executing a replay attack must be done before the actual response is sent from Alice. This is not possible, since the credentials have not yet been entered. The only way to authenticate without the password is with a MITM attack on both messages.

[17]Rather than being *logged in* to the system, one's actions are continually controlled by authentication, appending an `Authorization` header to every action. Credentials used for the response calculation are usually stored in the browser for the length of a browser session.

[18]Nonce lifetimes are implementation specific, set up during web server configuration.

Figure 2.8: A replay attack

This nonce calculation procedure is suggested in RFC 2617:

nonce = time-stamp H(time-stamp ":" ETag ":" private-key) [9, p. 8]

Where `H` is some hash function, `time-stamp` is the current time at the server, `ETag` is a *fingerprint value*[19] identifying the contents at the location and `private-key` is a value private to the server. Note that this private key is not used for encryption, but merely as an identification piece; it can be a random value generated by the server.

The values contained in the suggested nonce calculation are not thoughtlessly chosen. The time stamp separates each authentication attempt from every other, in time, and the ETag is used to prevent replay attacks against updated or dynamic pages. Finally, the private key is used for the server to ascertain that the integrity of the nonce does not change during transfer. Albeit being just a suggestion, the nonce calculation function above (at least the "time-stamp and hash" format) seems to have become the de facto standard.

Client nonce

The client nonce is a nonce passed from the client to the server to get the same assurance in the other direction as well. It is used only in conjunction with the quality of protection (qop) directive. We describe this below.

2.5.2 Quality of Protection

Quality of Protection, defined in the `qop` field of the `Authorization` header, describes what level of protection of the digest authentication scheme is used. It is sent from the server as an advertisement of available protection services. The client then selects one and returns this in its `qop` field. `qop` is defined for the following values:

auth

Used to authenticate the user to the server with a username and password. This is the original digest authentication protection level.

auth-int

As above, with message authentication (additional integrity protection): A hash of the entity body[20] is hashed into the final response value.

— (not set)

Defines degradation to RFC2069, the initial digest specification, to which RFC 2617 is backwards compatible. `qop` is not a requirement but a strong

[19] Fingerprinting is done by lightweight hashing algorithms, whose use is for distinguishing a value from another rather than authentication.

[20] The textual contents of the document loaded.

recommendation in Digest authentication. An undefined `qop` changes the way the response value is calculated.

This report will focus only on the normal cases of digest authentication, i.e. `qop = auth`, thus there will be no examples of additional integrity protection or RFC2069 compatibility.

(other)

Custom quality of protection values are also allowed in HTTP Digest Access Authentication. This way, if both server and client implement the same quality of protection, customized authentication schemes may be developed using the digest authentication framework.

```
1  foo:protected:0ba42b54aadec573fee155ecd67d2dbe
2  bar:protected:6023ec4af9e9edc3fa072bb540db5dc2
```

Format: `[username]":"[realm]":"MD5([username]":"[realm]":"[password])`

Figure 2.9: A sample ACL

2.5.3 Access Control List

The access control policies that are required for digest authentication must in each entry contain the username and realm, in addition to the password, or some means of establishing the veracity of the user credentials and thereby authenticating the user. In the Apache Web Server, which we use for various examples, the access control policy is a hidden file[21] referenced from a configuration file. In figure 2.9 we present a sample access control list (ACL) in the format username:realm:MD5(username:realm:password). The password is not written in clear text, but is hidden inside a hash. This way, the only way to validate a correct password is to calculate the hash of the same values.

2.5.4 Calculating the response value

The `response` value is calculated by the user agent as an answer to the server's authentication challenge (a `WWW-Authenticate` HTTP header). The `response` value is the result of hashing two independent parts. The first, called `HA1` is a hash of the realm and the user's credentials. The second, `HA2`, is a hash of the HTTP request method (GET, POST, etc.) and the URI of the target resource, thus different for every page. One can distinguish `HA1` and `HA2` as the secret and non-secret values, or as the static and dynamic components of `response`, respectively. The static component is the one stored in the ACL at the server side and is used in calculations to produce the response value on either side for comparison and validation at the

[21] Apache defaults to denying access to filenames with the `.ht` prefix. As a consequence, configuration files (e.g. `.htaccess`) and access control lists (e.g. `.htdigest`) cannot be fetched through the Apache web server.

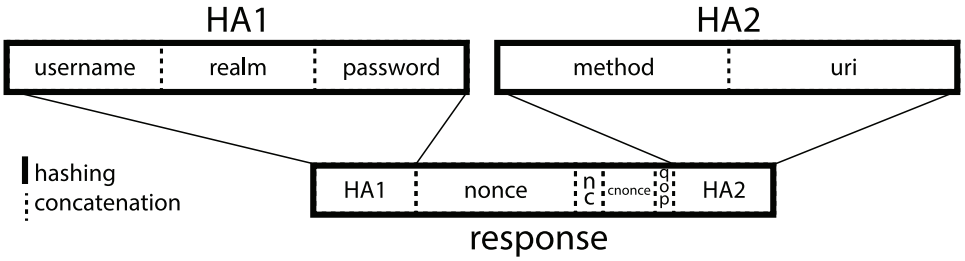


Figure 2.10: The modular architecture of the response value (preimage)

server. It should be considered "secret" because it contains the credentials of the user, and if compromised it can be replayed to the server. The dynamic component changes whenever the resource URI or HTTP method changes, and contains no secret information.

HA1

The contents of HA1 are dependent on the algorithm provided. If `MD5` or no value is defined in the `algorithm` field, only the user credentials and the realm are used:

HA1 = MD5(username:realm:password) [9, p. 12]

The realm is appended as a *salt* to the username and password. If several of the user's HA1 values are compromised and all use the same username and password, each different realm's HA1 value will be different – there is no way to directly infer relation between the user's identities on different realms. If the server requires HA1 to have time limited validity (`algorithm` == "MD5-sess"[22] or similar), nonce and cnonce (client nonce)[23] values are also applied:

HA1 = MD5(MD5(username:realm:password):nonce:cnonce)[9, p. 12]

Notice that the time limited HA1 value contains the original HA1 value. The lifetime of the MD5-sess version of HA1 is dependent on the lifetime of the nonces, as these are wrapped in. As noted in section 1.4.1, in this report we will focus only on the normal cases of digest authentication. We consider `MD5-sess` outside the scope of the thesis. Seeing that the HA1 from MD5-sess contains the HA1 from original MD5 authentication, both methods should be usable with the OffPAD using the same stored credential values.

HA2

When calculating the HA2 value, the `qop` field is taken into consideration. HA2 is defined for `qop`-values "auth" and "auth-int" and non-present `qop`

[22] MD5-sess works per-**sess**ion

[23] The client nonce is presented in section 2.5.8.

for backwards compatibility with RFC 2069. If `qop` == "auth" or non-present:

HA2 = MD5(method:uri) [9, p. 13]

Where `method` is the HTTP request method that was used. It can be any of the methods described in the HTTP standards, such as GET, POST, etc. All HTTP Request methods are defined in the HTTP 1.1 RFC [6]. `uri` is the URI of the resource that is to be retrieved. If `qop` == "auth-int":

HA2 = MD5(method:uri:MD5(entity-body)) [9, p. 13]

response

When both HA1 and HA2 have been calculated, the response value is finalized depending on whether qop is set: If `qop` is set (either `qop` == "auth" or `qop` == "auth-int")

response = MD5(HA1:nonce:nc:cnonce:qop:HA2) [9, p. 13]

Where `nc` is an eight hexadecimal digit counter incremented on each HTTP request. This way, the server could deny any request with an already used combination of `nc` and `nonce` as a replay attempt. If qop is not set, the response value must be backwards compatible with RFC 2069 using only the nonce value:

response = MD5(HA1:nonce:HA2) [9, p. 13]

A note on notation

`HA1` is the **H**ashed `A1` value. Thus, `A`x and `HA`x describe a value and that value hashed, respectively.

2.5.5 The state of MD5 and its use in Digest authentication

In Digest Access Authentication, the hash function MD5 [31] is suggested as the one-way function for creating nonces and calculating the digest of the user's credentials.

Since it was published by Ronald Rivest in 1992, MD5 has been subject to intensive robustness testing. At the 2005 Eurocrypt conference, Xiaoyun Wang and Hongbo Yu presented an attack that found a MD5 collision in between 15 to 60 minutes[24] [42]. This lead to Rivest (informally) announcing MD5 as *broken* [32]. However, this only applies to random and chosen-prefix collisions[25]; at the time of writing there are no practical attacks on preimage or second preimage. The best theoretical approach for a preimage attack is, at the time of writing, of complexity $2^{123.4}$ and was

[24] In 2009, Xie and Feng presented "an improved algorithm capable of generating a collision within one minute on a desktop PC" [44].

[25] Stevens et al. presented an efficient chosen-prefix attack on MD5 in [36] which is usable to create rogue X.509 certificates, such as those presented in figure 2.6.

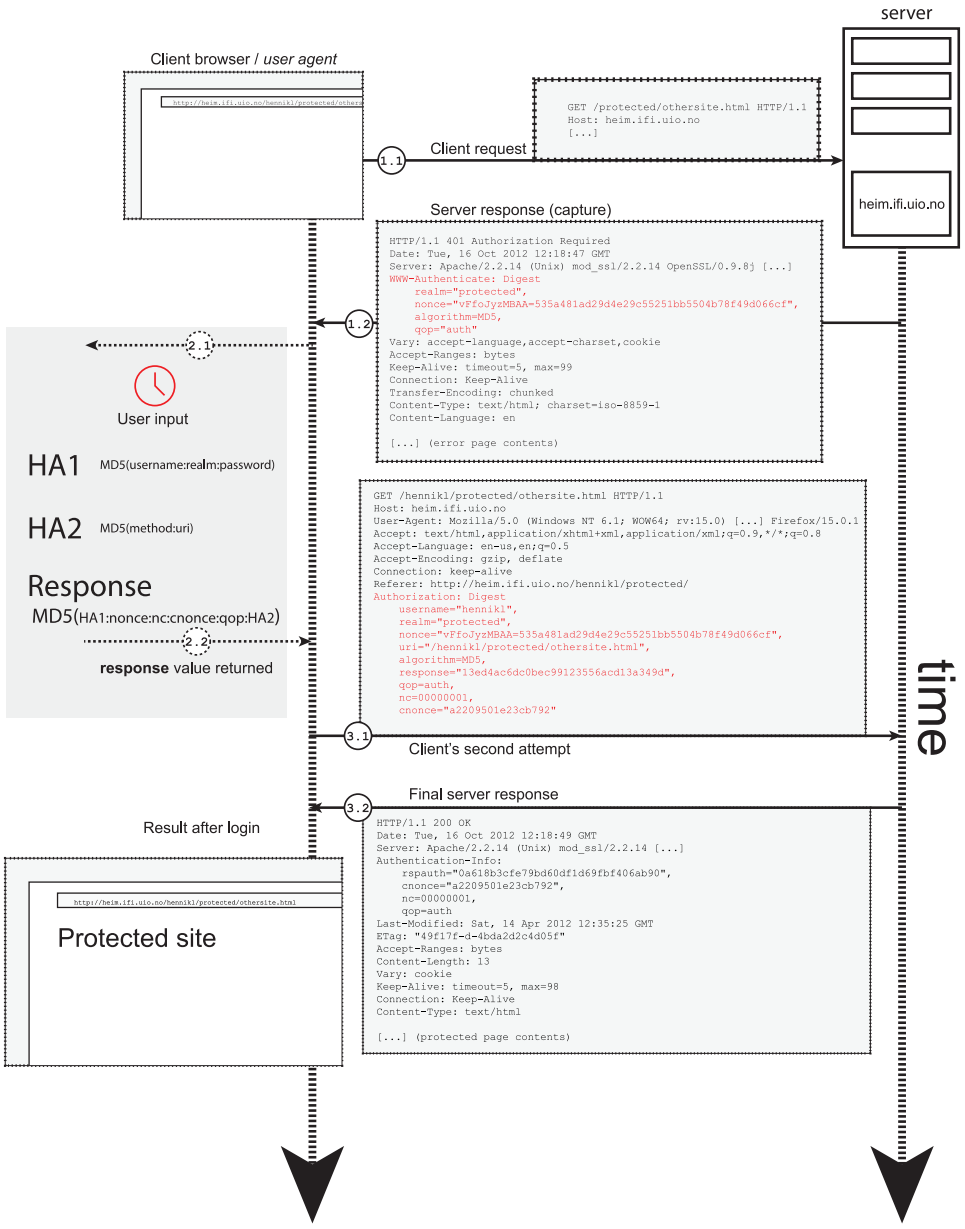


Figure 2.11: HTTP Digest Access Authentication

presented at the 2009 Eurocrypt conference by Yu Sasaki and Kazumaro Aoki [33].

The *broken* state of MD5 does not directly compromise its use in the communication of Digest authentication. If we assume that the server is authenticated and secure, there is no way an outside source can influence the nonce generation, and preimage attacks on MD5 hash are as mentioned only theoretical; the easiest way to get falsely authenticated is to perform an exhaustive search or dictionary attack on the `HA1` and `response` values, yielding the password. We show such an attack in section 5.2. However, note that RFC 2617 mentions MD5 only as an example, not a requirement. As long as both the server and client supports digest authentication in the specified hash suite, any suite is applicable.

2.5.6 Whether or not to use passwords

Storing the `HA1` value in the OffPAD rather than in the user's brain brings a huge benefit. There is no longer a need for a password. Removing the notion of a password, however, converts the authentication factor which was previously "something you know" (the password) into "something you hold" (the OffPAD device). Passwords can be replaced by random values that exhaust the maximum entropy of the employed cryptographic function. This means that using the MD5 hash function, we only need above 128 random password bits.

We benefit from the choice of such a random password in that there is no straightforward way to stage a brute force attack against a 128 bit random value. Human chosen passwords tend to be of low quality and one may even use side-channel information to stage more effective brute force attacks[14]. If at the same time the passwords are protected with a fast hash function, brute force attacks are even more effective.

2.5.7 The state of HTTP Digest Access Authentication

The current Digest authentication scheme is vulnerable to man in the middle attacks (MITM), where a false remote server may advertise that they only support earlier, vulnerable editions of HTTP authentication, such as RFC2069 and even Basic Authentication. We regard our implementation as secured against attacks of this nature as we have decided not to support the earlier schemes or backwards compatibility.

This leaves a scenario where a MITM or compromised server advertises a usable nonce and a valid realm for the user to authenticate to. The victim produces a `response` value based on the nonce issued by the false server that is fully usable. This attack is called a *chosen plaintext attack*, as the attacker is able to choose a nonce value. The nonce value is easily availed by the compromised server upon request, and there is no tie between a nonce and a user. Thus, with a chosen plaintext MITM attack, the user can surf the protected web pages and have its traffic monitored. However, intercepting only the `response` value from the victim (he never reveals his

password) the attacker cannot simply impersonate the user and decide on what protected data to browse on the remote system.

It is required that the attacker is in possession of the static credential data (`HA1`) or the user's password to authenticate as him. The only possibility an attacker has for recovering one of these from the `response` value is a brute force attack, to which HTTP Digest Access Authentication still is vulnerable. It is very important that none of the communicated data in the authentication scheme is decipherable or in any way can be used by an attacker. In section 5.2 we show how this holds if we use either random passwords or a key derivation function.

RFC2617 states that "The intent [of HTTP Digest Access Authentication] is simply to create an access authentication method that avoids the most serious flaws of Basic authentication." [9]. That may indicate that it is a "better than Basic but nothing more" solution. We decide to disregard this, as we see the scheme fit for the future with only minor modifications. In chapter 5 we scrutinize our scheme and suggest improvements, countermeasures and best practices.

Based on the above and from what we show in chapter 5, we must conclude that HTTP Digest Access Authentication is not secure enough as a contemporary authentication mechanism. Without the use of either long, random passwords or computationally intensive key derivation functions, it is not able to keep the privacy of users' passwords.

2.5.8 The HTTP Digest Authentication protocol in detail

In this section, we will go through all related HTTP status codes, headers and their fields.

HTTP Headers

`WWW-Authenticate` – Always present in a `401 Authorization Required` challenge. Its contents are meant to help the user decide on what credentials to provide and to help the client system (the user agent) calculate a response value according to protocol.

`domain` – A list of URIs to which the credentials are applicable.

`realm` – The name of the protected resource or directory presented to the user so that he understands what credential set to use. The realm name is the only value that separates one protection realm from another. Therefore, no two realms in one system may have the same name.

`nonce` – A challenge from the server. The nonce can for example be a hash of server-specific so that the origin of the challenge can be verified on the server.

`algorithm` – The digest algorithm suite used to create the response value. If not specified, MD5 is used.

`qop` – The Quality of Protection describes what protection services are supported by the service. Choices are `auth` (authentication), and `auth-int` (authentication with integrity protection), where `auth` is the default.

`stale` – The received nonce value does not match the value in the previous challenge or the nonce has timed out. In both cases, the `stale`-flag indicates that the response to the challenge is wrong and that the user is not authenticated, regardless of correct credentials and authorization. A fresh nonce is contained in the header when the stale flag is set.

`opaque` – A string generated by the server that is to be returned unchanged when navigating inside the protection realm.

`Authorization` – The Authorization header is the client's response to a server challenge originating from a `401 Authorization Required` status code. Authorization headers are passed with the `HTTP` request method and contains the client's proof of identity.

`username` – The username of the user.

`realm` – Same as above.

`nonce` – Same as above, returned to provide origin authentication of the challenge.

`uri` – The URI of the requested resource, to which the response applies.

`algorithm` – The *one* selected algorithm suite of the choice presented by the server in the challenge.

`cnonce` – If the qop field is defined in the challenge, the user must provide a client nonce. The cnonce is similar to the server nonce in that it identifies the server's knowledge of the client thus providing mutual authentication of the communicating entities. Also, the cnonce acts as a salt to the response value, making it impossible to build rainbow tables[26] for use in an attack scenario where the attacker controls the nonce.

`qop` – The selected quality of protection of the choices presented by the server in the challenge. An algorithm suite may specify different one-way functions protecting each of HA1, HA2 and response, or structure the response value another way.

`nc` – If the qop field is defined in the challenge, the nonce counter (nc) must also be present. The nc value distinguishes the current request attempt's response value from the other response values. This contributes to the countering of replayed messages.

`opaque` – The opaque value from the server, returned unchanged.

[26]Rainbow tables are pre-calculated tables of a large set of preimages and their corresponding hash value.

`response` – The calculation done by the algorithm defined in the algorithm field, which proves that the user has knowledge of his password. A detailed description of the response value calculation is shown in section 2.5.4.

`Authentication-Info` is passed to the client attached to the `200 OK` status message indicating that the authentication of the user was successful and that the user is authorized for accessing the realm. Following the headers is the contents of the protected web page. The Authentication-Info header, as the name suggests, provides the client with additional information about the authentication.

`nextnonce` – nextnonce is mandatory to use per policy, but is not used in the current Apache web server. It is intended to establish the nonce before nonce timeout, so that a stale-flagged message is not necessary.

`rspauth` – The response authentication value provides mutual authentication. By calculating rspauth, the user can ascertain that the server has knowledge of the user's secret credentials. The only difference between response and rspauth is the use of cnonce instead of the nonce.

`cnonce` – The cnonce originating from the client.

`nc` – As in Authorization.

`qop` – As in Authorization.

HTTP Status Codes

`200 OK` – Whenever a HTTP request is done successfully, a `200 OK` status code is returned, indicating to the browser that the page loading was a success. After all the headers, and broken by a blank line, the contents of the web page follow.

`401 Authorization Required` – When the web browser is navigated to a web page protected by a HTTP access authentication scheme, a `401 Authorization Required` status code is returned. The 401 status automatically invokes a credential input screen in the web browser for authentication.

2.6 Apache HTTP Server

For setting up a sufficient testbed for HTTP access authentication, Apache HTTP Server (or *httpd*) is a good option. It supports both the Basic and Digest authentication schemes. Below we describe two ways of configuring httpd to protect the contents of a server directory. We assume httpd is already installed on the server system[27].

[27] Apache HTTP Server is free open-source software, and can be retrieved from http://httpd.apache.org/.

Configuring httpd

Access authentication protection in httpd is done on directories and their contents and subdirectories. We can create an access policy by either editing the global Apache configuration file (apache2.conf or httpd.conf) or by creating a hidden file inside each protected root directory. Figure 2.12 shows a snippet of an httpd configuration file where the directory `protected` in `/var/vww/` is named `realmname` is protected with Digest authentication.

```
1  <Directory /var/www/protected>
2  AuthType Digest
3  AuthName realmname
4  AuthUserFile /var/www/protected/.htdigest
5  #AuthGroupFile /var/www/protected/.htgroups
6
7  require valid-user
8  #require group authorizedpeople
9  #require user hennikl
10 </Directory>
```

Figure 2.12: Digest Authentication on Apache Web Server

`#` – Lines that begin with hash symbols are comments.
`<Directory>` – Describes what directory is to be protected. The Directory enclosure is only used in the global configuration file. When setting up access authentication in a single directory using an `.htaccess` file, the directory is implicit.
`AuthType` – The type of authentication, either Basic or Digest.
`AuthName` – The name of the protection realm.
`AuthUserFile` – The absolute location of the user file (explained below).
`AuthGroupFile` – The absolute location of the group file (explained below).
`require` [user | group | valid-user] – Describes what users or user groups from the digest file are allowed access to the realm. valid-user describes any authorized user from the user file.

The group and user files

The user file `AuthUserFile` is the ACL in Apache Web Server's digest authentication. When the server receives an authentication reply from the client, it checks the username against the configuration. The parameter `require` decides what users or user groups are authorized. User groups are written on the form [group] [user 1] ... [user N]. The format of the user file can be seen in figure 2.9.

Chapter 3

General discussion

3.1 Evaluation of the OffPAD

In this section we evaluate the OffPAD design and the contained technologies following the framework specified by Bonneau et al. in *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes* [5]. The 2012 report provides a total of 25 "benefits" separated into the classes "usability", "deployability" and "security". An extensive list of different online authentication schemes is evaluated[1] and compared with regard to the aforementioned benefit classes. We evaluate our scheme this way for it to be easily comparable with other similar schemes in the future.

For the below evaluation, section II, "Benefits" of [5] should be used for reference. We evaluate the OffPAD based on these benefit classes using the original names and definitions from the report.

3.1.1 Usability benefits

For a new device[2], usability is critical if it is to be adopted on a large scale. An authentication scheme does not sufficiently benefit its target audience if it cannot be easily used. Therefore, Bonneau et al. propose the following usability benefits for authentication schemes:

Memorywise-Effortless

While a future OffPAD may embed biometric access control to the device, by the current OffPAD design it is required that the user remembers a single device master password. Therefore it is *Quasi*-Memorywise-Effortless.

Scalable-for-Users

As it is able to contain a large amount of online identities, the OffPAD is Scalable-for-Users. The only limit to scalability is the size of its on-board persistent memory.

[1] In total 38 schemes are evaluated, where 36 appear in the comparison table [5, Table 1]

[2] New in the sense that it is not yet widely deployed in the market.

Nothing-to-Carry

If the OffPAD is implemented as part of a mobile phone (which we pointed out in section 2.2.4 may be a bad idea) it is Nothing-to-Carry, as the mobile phone is something that is carried regardless: having an OffPAD inside the phone will not further encumber the user. However, for now we consider the OffPAD to be a standalone device, consequentially it is not Nothing-to-Carry.

Physically-Effortless

There is some physical effort required for handling the OffPAD, closely tied to the security and the *mostly offline* requirement of the device. For it to stay disconnected from the client computer, the device must be physically moved into the communication range of the client computer for every authentication. As this interaction is slightly more demanding than "pushing a button", as is the example of Physically-Effortless in [5], we deem the OffPAD not Physically-Effortless.

Easy-to-Learn

Considering the authentication phase, moving the OffPAD within the communication range of the computer should not be hard for the user to understand. In the registration phase, the creation and persistence of credentials may be somewhat difficult. The totality of learning does not range beyond the notion of Easy-to-Learn. Therefore, the OffPAD is Easy-to-Learn.

Efficient-to-Use

Both the time it takes to register a credential set on the device and the time it takes for each authentication to take place is significantly reduced by the OffPAD compared to the original scheme. It is Efficient-to-Use.

Infrequent-Errors

Relying on the quality of the credentials stored on the OffPAD, it has no reason to err, thus has Infrequent-Errors.

Easy-Recovery-from-Loss

Should the OffPAD get lost or stolen, its contained credentials follow. The current idea of an OffPAD does not contain a formal definition of a backup solution, but one could be easily implementable. Such a backup could be stored either on the client machine or in some central location ("http://offpad.com"). The `HA1` values must be protected, perhaps encrypted with a key derived from the device's master password.

An eventual backup service is future work, so the OffPAD cannot be classified Easy-Recovery-from-Loss.

3.1.2 Deployability benefits

The deployability benefits reflect how well the scheme fits in as a newcomer in the "authentication scheme flora". They may indicate how likely the scheme is to be adopted by end-users over the other competing schemes.

Accessible

The OffPAD is accessible to anyone who is capable of performing regular password authentication. Indeed, the scheme is accessible to anyone who is able to physically move a device (initiate a connection). We deem the OffPAD Accessible.

Negligible-Cost-per-User

With any device follows a cost. While service providers may issue their customers free OffPADs, there are no scenarios where an OffPAD can be free at all levels, and therefore cannot be considered Negligible-Cost-per-User.

Server-Compatible

We describe in chapter 5 that the OffPAD has several optional security mechanisms to future-proof its operation. While the OffPAD's authentication mechanism builds on the original HTTP Digest Access Authentication scheme, we show that the current state of digest authentication is insecure. Two examples show how the identities can be stored safely on both the OffPAD and the server.

(a) If using the original scheme where the credentials are stored as uniterated MD5 hash values, the plain text password itself must be of sufficient entropy to evade brute-force attacks. This means that the password's entropy must be on or above that of a random 128 bit value. As MD5 is still preimage resistant, there is no way to recover a random, MD5-hashed 128 bit value, even if protected only by a single iteration of MD5. Thus, having the OffPAD or the remote identity management system generate a 128 bit random value as the password, the original digest scheme is usable, and the OffPAD is Server-Compatible

(b) If the remote system for some reason requires passwords that can be remembered by the user, a password protecting mechanism should be introduced, such as a key derivation function (see section 2.3.1). Key derivation functions run one-way functions with an appended salt, through thousands of iterations, producing a strengthened hash, on which a brute force attack will take longer per guess (see section 2.3.1)[3]. Some password policies restrict the length of the password

[3]However slow, KDFs do not guarantee for the security of the password, only additional protection.

and the available character set. Representing the 128 bit value as 16 bytes may be compliant with the length rule. Representing the values as 32 hexadecimals will be compliant with the character set rule. Regardless, one can see that these tricks cannot easily be used together.

Solution (b) is not Server-Compatible, as it requires changes to the implementation of digest authentication (a custom quality of protection mechanism[4]), but is only provided as an addition to the OffPAD concept. As (a) is usable in most scenarios we regard the scheme as Server-Compatible.

Browser-Compatible

The OffPAD is not Browser-Compatible as it requires a browser extension or client software on the machine to mediate message passing between the OffPAD and the server. Browser compatibility is also broken in order to break the browser's authentication context.

Mature

As the concept has not yet been deployed for or tested on a large audience, the OffPAD and its underlying technologies are not Mature.

Non-Proprietary

The extended digest authentication scheme which runs on the OffPAD is patented, and therefore the solution is not Non-Proprietary.

3.1.3 Security benefits

A third important factor in an authentication scheme is its ability to protect the authentication transactions from adversaries, who are either trying to collect the credentials or replay authentication.

Resilient-to-Physical-Observation

No data that is critical to authentication is ever shown on the OffPAD's screen, or on the client computer. The OffPAD is Resilient-To-Physical-Observation

Resilient-to-Targeted-Impersonation

The OffPAD's credentials are stored on the device, and must be unlocked by a master password to become active. Assuming that the master password cannot be deduced from personal information, the OffPAD is Resilient-to-Targeted-Impersonation.

[4]See (other) in 2.5.2

Resilient-to-Throttled-Guessing

As described above, relying on the preimage of the employed one-way function makes the benefit Resilient-to-Throttled-Guessing irrelevant in this context; there is no reason to throttle a brute force attack of complexity $2^{123.4}$ [33]. If this security control is required, implementation is trivial. In both cases, the OffPAD is unconditionally Resilient-to-Throttled-Guessing.

Resilient-to-Unthrottled-Guessing

Again, as previously described, the OffPAD is Resilient-to-Unthrottled-Guessing and Resilient-to-Throttled-Guessing regardless of specific security controls. All critical information passed between the OffPAD and the server consists of a pseudorandom credential that are protected by a one-way function, or a password protected by a KDF. These protection mechanisms make the OffPAD resilient.

Resilient-to-Internal-Observation

If it is possible to gain access to the OffPAD's memory whenever a `HA1` value is present, it can be collected from memory. An `HA1` value can be used (replayed) to authenticate the registered user to the realm specified in the `HA1`. In order to retrieve the password, however, the `A1` value must be recovered and the `username` and `realm` removed[5]. The OffPAD is not Resilient-to-Internal-Observation.

Resilient-to-Leaks-from-Other-Verifiers

This requirement is also dependent on the server's protection of the `HA1` values. If protected by a KDF or using randomized passwords to generate `HA1`, an exhaustive search within `response` would be infeasible[6]. Compromise of the server will still protect the credentials sufficiently to be Resilient-to-Leaks-from-Other-Verifiers. If today's digest authentication scheme is used, however, guessing a password from a *snooped* `response` value is only twice as hard as *cracking* an MD5 hash (see section 5.2).

Resilient-to-Phishing

Server-side authentication is outside the scope of this thesis. If the OffPAD is configured with a Petname system for phishing protection, as described in [41], the OffPAD will protect its user from authenticating to sites that claim to have a registered user in the OffPAD. Relying on this solution, we assume the OffPAD also to be Resilient-to-Phishing.

[5]The password itself is only more valuable than `HA1` in situations where the same password has been used for several of the user's accounts. If following the best practices laid out in this report, each password for every service is random.

[6]Assuming, of course, that the password in the KDF case is of somewhat high entropy.

Resilient-to-Theft

Being a physical unit, the OffPAD cannot be resilient to theft. However, the wording in the framework by Bonneau et al. suggests that the resilience to theft is a measure of the *consequence* of having one's device stolen is, regarding misuse and false authentication with the device. Being protected by a master password, the OffPAD qualifies as Resilient-to-Theft.

No-Trusted-Third-Party

In a future version including a backup solution, the backup server may be seen as a trusted third party in the OffPAD communication. The current OffPAD has No-Trusted-Third-Party.

Requiring-Explicit-Consent

Authenticating with the OffPAD is activated by Requiring-Explicit-Consent, through the act of physically moving the device within the communication range of the client computer.

Unlinkable

There is no explicit link between a specific OffPAD device and the usernames registered on it. The device never reveals (through communication) a device ID or any similar information that is may reveal the identity of the user. However, of course, usernames and realm names may be transmitted in clear over HTTP and are not considered sensitive. Colluding verifiers (identity providers) may conclude that several users are linked if the identities are registered with the same, or similar usernames on each service. The definition describes being Unlinkable as not being able to draw this conclusion from the authenticator alone. This is not possible *per se*, but because it likely that an OffPAD owner will register with similar usernames to different services, we deem the OffPAD *Quasi*-Unlinkable.

3.2 The OffPAD in contrast to other authentication mechanisms

Here we present our investigations of how the OffPAD design differs from a couple of similar contemporary proposals. These were introduced and described in sections 1.5.1 and 1.5.2. One scheme whose design is very similar to the OffPAD's is the Pico. It is envisioned by its inventor Frank Stajano as a "usable and secure memory prosthesis" [35], and with that shares a common goal with the OffPAD. Further, in the position paper *Choose the Red Pill and the Blue Pill*, Singer and Laurie propose a design of a "small handheld device, with a graphic user interface, manual input [...], one or more communications interfaces[...]". This device is supposed to work as an additional operating system side by side with the assumed

	Pico	OffPAD
Communication	Visual code (camera facilitated) for challenge, wireless communication for response	NFC or other physically activated
Protection	Asymmetric cryptography	One-way function (hash / KDF)
Physical design	Small device, ideally kept within another unit	Small standalone device or part of mobile phone
Peripherals	Camera, wireless	NFC
Number of devices needed	Multiple *picosiblings*	One
Requires changes to server	Yes	No
Requires changes to client	Yes	Yes
Scope of operation	Applicable to all systems	Computer system authentication
Backup solution	Yes	No

Table 3.1: Comparison between the Pico and the OffPAD

vulnerable client system, which is also a goal of the OffPAD. Because "one simply cannot properly secure a general-purpose operating system" [23].

3.2.1 The Pico

The Pico design is a *yet-to-be-implemented* idea of an authentication device for every person to carry around. In its design, it is suggested that each person carries a Pico with several *Picosiblings* (more Pico devices) around. These will intercommunicate their presence and provide high availability. Such a pico-network[7] provides the Pico owner with *personal mobility* of his authentication service. The scheme benefits from this redundancy: should one Picosibling disappear from the network, there is always another, keeping the system available. However, the Pico design suffers from poor usability when consisting of several units. Stajano suggests that this might be alleviated by embedding the Pico functionality in a mobile phone, or other units that people already carry, that do not further encumber the user. He imagines Picos in the shape of "a watch, a key fob, a bracelet or an item of jewellery." [35]. The OffPAD has no such demands for high availability.

The proposition demands, however, as we quoted in section 1.5.1, that there are no application restrictions to the Pico scheme. This means that all remote access control systems must adjust their authentication framework in order to be eligible for Pico authentication, a radical demand for an authentication scheme. The Pico covers the part of the OffPAD design that deals with identity management and automatic authentication, but has a more narrow focus on targeted systems.

[7]Not to be confused with a *piconet*, which is a network of interconnected Bluetooth devices.

3.2.2 Nebuchadnezzar

As we discuss in section 1.5.2, the Nebuchadnezzar design (i.e. the proposed external device) does not differ greatly from the OffPAD. In fact, one may see the OffPAD as a physical implementation of the abstract design. There are, however, some important differences in the motivation and use of the Nebuchadnezzar. The below are commented excerpts from the Nebuchadnezzar specification:

1. "We propose using the general-purpose operating system for everything but the bits that need security" [23]
   – Where the Nebuchadnezzar may be used as a general all-encompassing facilitator of security services, the OffPAD is specifically concerned with user authentication only.

2. "We do admit that it might be possible, with sufficient care, to somehow share the same device as a mobile phone and as a Neb." [23]
   – As we argue in section 2.2.5, using a mobile phone as the OffPAD is incompatible with our assumption of the mobile phone as a general purpose (hence unsecurable [23]) operating system. We therefore require the mobile phone, if used as an OffPAD, to be a separate entity from the phone itself, and run on its own physically disconnected operating system. This way, both the OffPAD and the phone will not share devices, but be able to interact with each other. The above statement does not conflict with the OffPAD design directly, but we require a stricter definition of how the functionality may be combined with a mobile phone.

3.3 TazCard as the OffPAD

In section 2.2.4 we described some special requirements for the OffPAD. It must be mostly disconnected, meaning that all possible contact is restricted, and that every connection requires consent in its physical activation. Furthermore it must be able to securely handle identity management. TazCard from the French company TazTag was a good fit for these requirements. TazCard is a security focused small device. With a wide support of connectivity and I/O, it can be customized for a wide range of applications. The TazCard has native support for Java MIDP[8] applications – so-called MIDlets – that can be deployed to the device via USB over SCP[9]. Application developers can utilize TazCard's connectivity API for USB, ZigBee and NFC communication.

[8] Mobile Information Device Profile
[9] SCP - Secure Copy is a file transfer protocol.

Connectivity

A very important requirement for the OffPAD is that it is sufficiently disconnected, or offline. NFC[10] is a communications technology that allows us to keep the OffPAD disconnected most of the time, as it requires the user to physically initiate a transfer. One must physically move the source (sender) device into the transmission range of the destination device to initiate a transfer. The TazCard has additional support for ZigBee and USB, from which we selected USB for testing purposes. ZigBee is not yet a widely accepted communication standard for PC communication, and is also a wider-range technology (10-100 m).

Identity storage

NFC is perfect for transferring identities to the device, by e.g. NDEF[11] formatted RFID cards. Support for this is implemented in the identity management application. It is possible that identities could be created on the computer and transferred via USB or ZigBee, or input directly on the device through its user interface.

A probable use case is having the identity provider issue an RFID card with the user's identity, and have the user transfer the identity to the OffPAD himself.

3.4 Authentication with the OffPAD

We can say that the `response` value in digest authentication is *modular*, in that it consists of several "layers" of values before it is finally hashed. From section 2.5.4 we saw that the `response` value is calculated using two already hashed modules, `HA1` and `HA2`, which we call the static and dynamic components. The `HA1` value is static since its result after hashing, called `A1` (`username:realm:password`), never change. `HA1` as a *credential hash* can be stored securely and used to create response values on every authentication upon request.

Remembering that the credentials, as used in the challenge-response communication of digest authentication, produced by a one-way function allows for confidential interchange of these credentials between endpoints. The confidentiality is provided in that we have a transient hash value (the `response` value) proving that the user is in *possession* of the correct credentials, rather than transferring the actual credentials. Since both endpoints are in possession of the credentials, proving possession is sufficient, and possible. This is the reason why the password does not have to be stored in clear text on the server. This proof of possession value (`HA1`) is stored both on the OffPAD and server, enabling us to require the `HA1` calculation, and the original password, only once – upon registration.

[10] Near Field Communication

[11] NDEF is the NFC Data Exchange Format – a formatting standard for RFID cards.

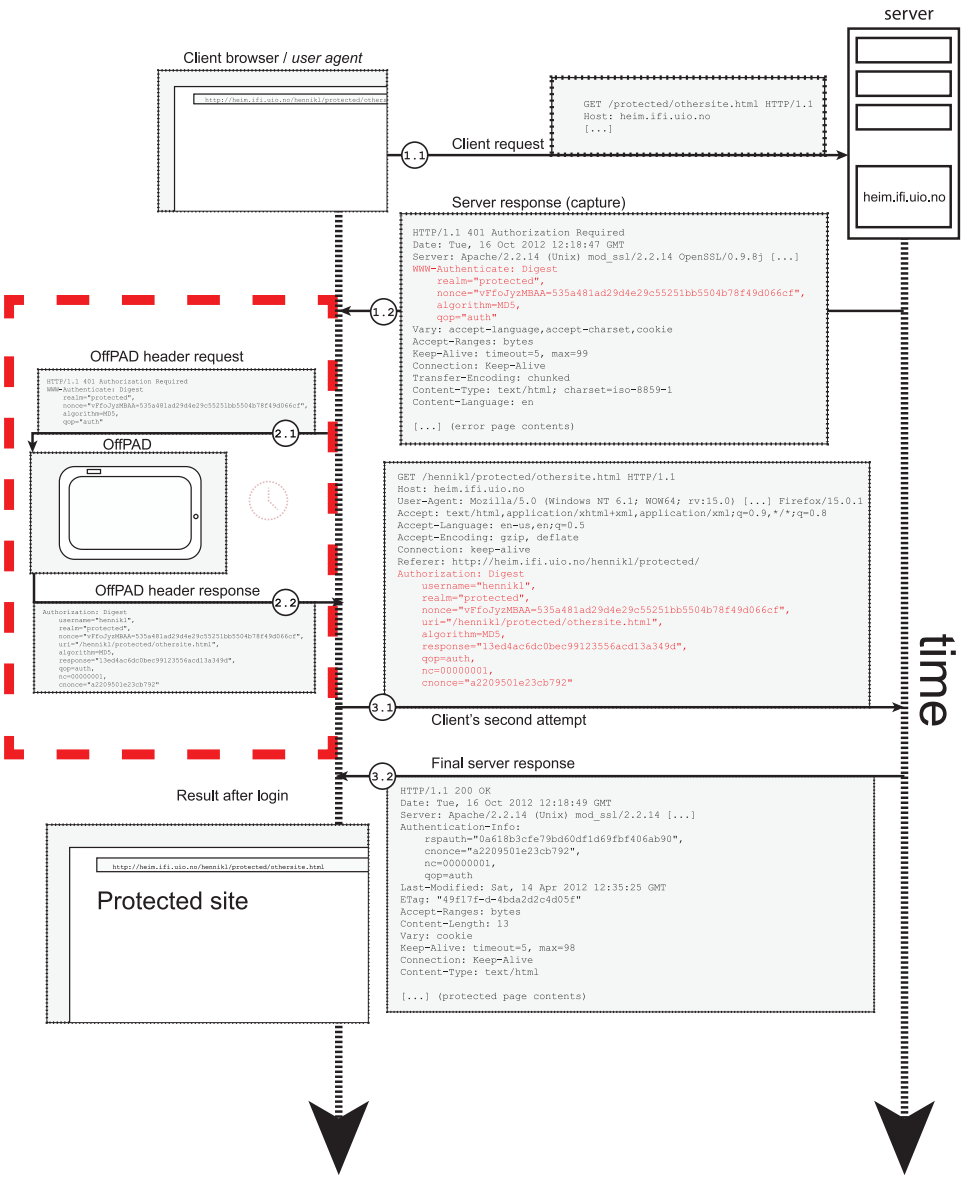


Figure 3.1: Extended HTTP Digest Access Authentication on the OffPAD

The client computer is an entrance for an attacker seeking to retrieve the credentials as they are entered or checked. Demoting the computer to an intermediary in the authentication communication, we transfer the proof of credentials between the server and the external secure unit (the OffPAD). Storing the `HA1` value on the OffPAD, the OffPAD handles the response value generation and credential selection based on the `HA1` value alone – there is no need to introduce the password outside of the registration phase. This way (again assuming that the server is secure) the user credentials never appear at a vulnerable location in clear text.

Passing the proof of identity from the OffPAD is done at *time of check*, i.e. at the moment credentials are requested by the user agent. Instead of waiting for manual input from the user, normal authentication flow is either split or intercepted. The `401 Authorization Required` status code mentioned above and the information it contains are rather passed to the device than the user.

The control flow of digest authentication using an OffPAD is described below. Compare this to the normal digest authentication control flow we showed previously, in 2.5.4. Figure 3.1 can be used as reference.

1.1 The client passes an HTTP GET request to the server at http://example.com/, requesting the /protected/ directory's index file.

1.2 A `401 Authorization Required` status code is sent back to the client, together with a `WWW-Authenticate` header containing information on the realm and what approach the user-agent should take to authenticate its user.

2.1 When the `WWW-Authenticate` header information arrives, the parts relevant to the authentication process (highlighted in the figure) are passed to the OffPAD via user-activated NFC. The problem of extracting header information from the browser context is discussed in section 3.7.

2.2 The OffPAD has a listening service already running. It notices the incoming information, and matches it against known identities (`HA1` values) for the requested realm, contained in the OffPAD. If no identity is found, the user is prompted for credentials and the resulting identity is stored for later. If multiple identities exist for the same realm, the user must decide on which one to use.

3.1 The hashed result of the selected local identity and the incoming header values are received by the computer and passed to the destination server authenticating the user. The authentication process will be completely transparent to the server (i.e. it will not differ from the normal case where a user enters his password).

3.2 The server responds with a `200 OK` status code and the contents of the protected resource the user requested.

The OffPAD as an identity manager requires that the identities are registered on the device, or can be loaded on request. Similar to a web

browser password manager such as Opera's[12], the OffPAD can store the values needed to create a valid hash for each web site, based on the input described in point 2.2 above.

3.4.1 Authenticating to the OffPAD

For the owner to unlock the OffPAD's identity management functionality he must authenticate himself by entering a PIN number or passphrase, or using another authentication method such as biometry. Authentication may be validated by comparing a hash of the provided authentication factor with a locally stored hash of the same value, created in the registration phase.

3.5 Identity management with the OffPAD

Critical to the overall security of a system is the protection of its access credentials. If a user's credential set is stolen, it only affects that user. Exploits are limited to his privileges[13]. However, if an identity store is compromised, the security of the contained identities relies on the quality of the mechanism protecting them. In the case of original HTTP Digest Access Authentication, the protection mechanism of the `HA1` value is a single iteration of MD5 on the concatenation of the credentials, which as discussed in section 2.3.1 is vulnerable to brute force calculation. For an identity store to be protected sufficiently we at least require the use of an iterated hash function, such as a key derivation function, or encryption of each identity. If the stored identities are to be encrypted, the key can be derived from a password, e.g. using the Password Based Key Derivation Function 2 (PBKDF2) [40]. Should an attacker be able to retrieve such an identity storage, he will not be able to recover plain text identities just as easily.

When user credentials are securely stored, both on the OffPAD and on the server, what remains is the need of an authentication framework that protects the credentials in transit. HTTP Digest Access Authentication transmits its proof of identity (the `response` value) to the server protected by the MD5 hash. We assume that the credential value (`HA1`) is sufficiently protected, as discussed above. The original digest authentication scheme is vulnerable in its use of single-iteration MD5, a weakness presented and discussed in section 5.2.

3.6 Server authentication

When the user navigates to a web site (e.g. `https://paypal.no/`) to which he wants to authenticate, he must be certain of the server's identity before providing his credentials. A malicious web page (e.g.

[12] http://help.opera.com/Windows/11.60/en/wand.html

[13] Of course, if the user has privileges beyond those of the other users (i.e. a super user), the consequences of a compromise are different.

`https://paypal.nu/`) may be used as a proxy, configured to collect the user's credentials before successfully authenticating the user to the correct site and redirecting him there. This is one of many *phishing* schemes, luring users into revealing private information. In the thesis work we have assumed that the server is the correct one, or already authenticated, e.g. by a correctly defined SSL certificate. The *petname* system in [41] is a possible provider of server authentication.

3.7 Collecting the Authentication header

When the user navigates to a protected web page that is supported by the OffPAD's authentication service, the user must be notified that he must move his OffPAD into the communication range of the computer to initiate authentication. The OffPAD must receive the header information from the request as quickly as possible, and use it to calculate the response value that proves the authenticity of the user. The problem of header collection can be solved in a number of ways, each being either an active or passive collection method.

Active collection breaks the original control flow of digest authentication. The headers are removed before they reach the browser, so the browser does not interpret the request for authentication, and the original "username and password" login window never appears at the computer.

Passive collection extracts the header without interfering with original control flow. Consequently, the user will still get the ability to authenticate manually from the browser, even if authentication has already taken place outside of the browser context. This approach may confuse the user, because the login window appears regardless of the success of the external authentication. Some possible collection methods are shown in table 3.2.

For the best usability, we require that the user is not presented with the login window unless the automatic authentication implementation fails – this to avoid confusion. We also require that as much as possible of user interaction with the computer is removed. We conclude from our suggestions, that the browser extension solution is the most reasonable one in that it is able to gracefully break the control flow and, relative to the alternatives, is very simple to set up. The browser extension software additionally benefits the user in that it does not require any particular learning by the user, and is rather unobtrusive. The extension is presented in section 4.2.

3.8 Software

The software discussed within this thesis includes the following three:

(a) HTTP Digest Handler MIDLet – A Java J2ME implementation of identity management and automatic HTTP Digest Authentication on a TazCard.

Method	Active/Passive	Advantages	Disadvantages
Firewall rule or socket-level redirection	A	Can filter out digest headers and route them directly to the OffPAD	Very difficult to implement – too advanced.
Browser extension	A	Can prevent the browser from loading the login window and let the OffPAD take over the authentication context	Somewhat difficult to implement
Packet capture	P	Can probe the client computer's network interface card (NIC) for incoming HTTP requests, easily filter digest headers and route them to the OffPAD	Does not interfere with the browser's login window causing confusion. Additionally, a large storage and processing overhead.

Table 3.2: Different approaches to collecting header information

(b) HTTP Digest Handler Application for Android – An Android application port of the above J2ME application.

(c) HTTP Authentication Extension for Google Chrome – A JavaScript-written extension for Google Chrome which passes authentication requests to the OffPAD application handling authentication.

Software implementations (a) and (c) have been successfully tested together, and provide the user with a functional automatic authentication service. Software (b) was not finished, lacking a proper test device (the TazTag TPH-ONE). All these are presented in chapter 4.

3.9 Use case scenarios

In this section we suggest a couple of use cases in which the use of the method is shown.

3.9.1 Registration phase

The registration phase for the extended digest authentication scheme is not significantly different from the original scheme. We assume that identities are stored in an access control list (ACL) at the server in the same way that they are stored on the OffPAD, as this facilitates synchronous calculation of the response value.

Premises:

- The user wants to register to a specific online service that requires HTTP Digest Access Authentication.
- The user is carrying an OffPAD with the HTTP Digest Handler application set up. The mechanism is configured in compliance with HTTP Digest Access Authentication.

Case:

1. The user is requested that his OffPAD is connected.
2. The user authenticates to the OffPAD, which unlocks upon correctly entered credentials.
3. The user transmits a unique identifier of the user, e.g. a username, the device's ID, etc.
4. The server registers the user identity and authorizes the user access to the service, by appending the identity to the access policy.
5. A password is chosen for the user's account. Since there is no longer a requirement that the user remembers his password, it should ideally be a long random number. From section 2.5.6, we draw the conclusion that a 128 bit random value is sufficient for our purposes.

Both the user and the server may generate this password, as long as it is delivered to the other party in confidence, so that it is not revealed, and the authentication process' calculations can be done synchronously. The `HA1` value is computed as

$$HA1 = H(username : realm : password)$$

where H is an agreed-upon digest function, such as SHA-256 or MD5.

6. `HA1` must be transferred to the other party in one of the following ways:
   (a) Securely online, e.g. by using asymmetric encryption or similar.
   (b) Transferred by mail (out of band) to the user, delivered by courier (signed, personal receipt), etc. This is necessary for high assurance level services, such as those provided by governments, etc.

   In the (b)-case we suggest using an RFID card as transportation medium. The OffPAD could support contactless communication such as NFC, enabling the reading of encrypted `HA1` values for local storage.

3.9.2 Online bank authentication with the OffPAD

Premises:

- An authentication server connected to the bank.
- A customer (user) carrying an OffPAD wants to authenticate to the bank.
- The user is authorized access to the bank.

Case:

1. The user navigates his web browser to a protected web service.
2. The user is requested that his OffPAD is connected.
3. The server replies with an authentication request, whose contents are routed to the OffPAD.
4. The OffPAD, using the `HA1` value and the incoming request parameters generates a response value, which is passed to the server in a response header.
5. The authentication server validates the identity of the user against the `HA1` value stored in its access control policy.
6. If the authenticated client is authorized for access (i.e. his `HA1` value is noted in the access policy), he is approved access to the service.

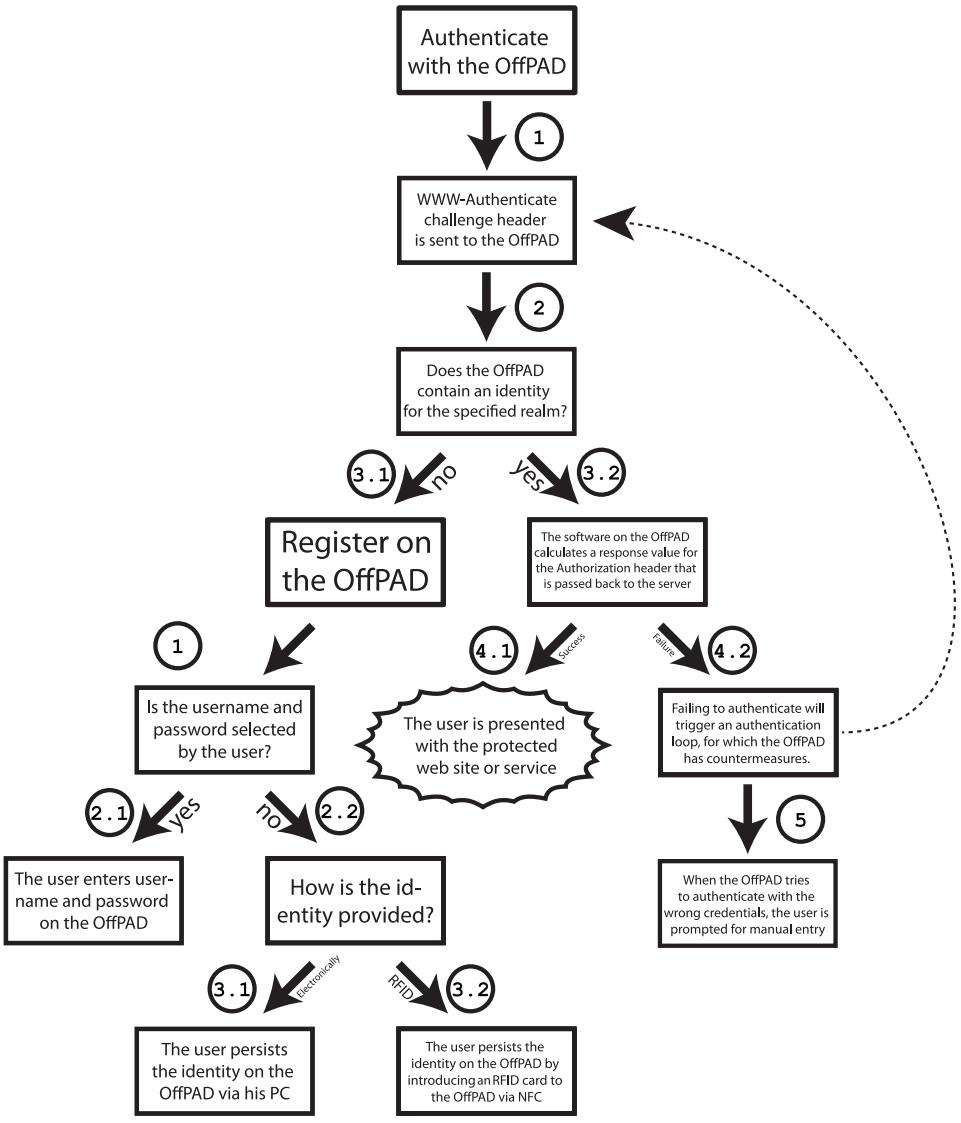


Figure 3.2: A flowchart describing the phases of operation

Chapter 4

Technical description

In this chapter we present the structure and flow of the software in detail, and how the theory from the earlier chapters is implemented as an OffPAD application.

There are two separate applications needed for the client computer and the OffPAD to pass authentication information. In the following we refer to these as the client- and authenticator software, respectively. The client software is called the *HTTP Authentication Extension for Google Chrome*, a browser extension handling incoming authentication requests, passing them to the OffPAD rather than the user. The authenticator software, *HTTP Digest Handler*, has been developed for different OffPAD platforms: a TazCard and a TazPhone (TPH-ONE) application. The latter has not been fully developed, as we lacked a proper testing device during implementation. The TazCard application is a J2ME based mobile application and the phone runs on the Android platform. Both applications communicate with a secure element embedded in the device. The authenticator software for the TazCard is presented immediately below, followed by a short comment on the TazPhone in section 4.1.4. The browser extension (client software) is described in section 4.2.

4.1 HTTP Digest Handler MIDlet

The main responsibilities of the OffPAD are identity management and to facilitate authentication using the managed identities. Its main task is to receive `WWW-Authenticate` header fields and in return create response header values based on the stored identities and the received data. Identities are stored on the TazCard's internal persistent memory. Using the `javax.microedition.contactless` library[1] to harness the TazCard's built-in NFC capabilities, it is possible to transfer identities to the TazCard from RFID cards. Identities are written to these RFID cards in the NDEF format. This is discussed in section 4.1.3. Any identity used is stored locally, if possible. For message passing to and from the computer, it is connected with USB. In a final version, the USB connection should be a secondary

[1] The J2ME Application Programming Interface for contactless communication.

(fail safe) option chosen over wireless communication (NFC). As argued in section 3.3. We suggest that only contactless communication is used, as it is the technology best suited for keeping the OffPAD "mostly offline". Contactless technologies require physical interaction, therefore conscious consent from the user, to function.

In order to recreate the TazCard execution examples and the use cases described in this report, a number of prerequisites apply:

1. A client computer system with the following:
   (a) A web browser with support for HTTP Digest Access Authentication. This is supported by every popular web browser.
   (b) A local network or Internet connection[2].
   (c) A USB port for communication with the OffPAD.
2. A server acting as authentication server. The server can be remote to the client machine, on the same network or run on the client machine itself[3]. It must have:
   (a) Support for HTTP Digest Access Authentication[4].
   (b) An access control policy[5].

In the following, the *client side* is understood as either 1. or the human user; based on context. *Server side* is as described in 2.

4.1.1 Architecture

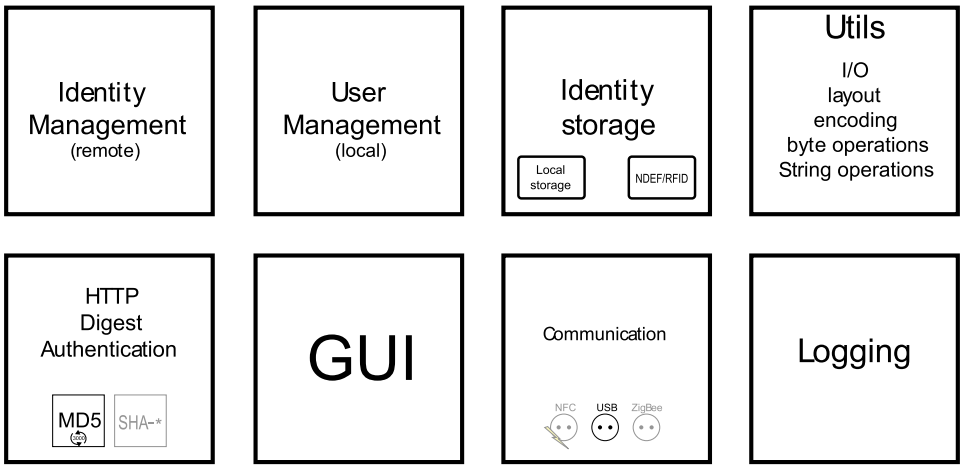


Figure 4.1: Architecture of the HTTP Digest Handler

[2]This is not required if the server is run locally on the client computer.

[3]Configuration of HTTP Digest Access Authentication on the Apache HTTP Server is described in 2.6.

[4]Supported by all popular servers, as it is part of the HTTP authentication framework.

[5]This is usually a list of authorized users and a digest of the username, password and protection realm.

The TazCard application is built up by modules, each with their own area of responsibility. Figure 4.1 shows the assembly of those modules, and can also be used as a guideline for the development of other OffPAD authentication applications. Suggestions / future possibilities are shown in grey.

Identity Management

The Identity Management module is the management system for remote identities, i.e. the online identities or *credential set* that the owner of the device assumes when using the device for authentication. This module handles the creation and modification of identities and works together with the Identity Storage module.

User Management

The User Management module manages all the OffPAD's local identities. This means that one authentication service may have several users on the same OffPAD. This is relevant for families or groups of people sharing one OffPAD. As an example, consider a workplace situation where several users interact with the same computer system. To authenticate, each employee can unlock their personal "OffPAD profile" on the shared device (with their personal master password) and authenticate throughout their session with the shared computer, until another user unlocks her profile. This way, each employee is relieved of bringing her own OffPAD to work.

Identity Storage

Identity Storage comprises all storing of remote identities. The application supports storing identities either locally on the device, or on external RFID identity cards. In some situations, one might benefit from decoupling the identity and authenticator, perhaps when authenticating from a shared, but trusted OffPAD, e.g. at a library[6], or again a workplace. If an authentication request appears on the OffPAD for a realm to which there are no available identities, it requests the user to register or provide temporary authentication. This can be done on the spot by moving an identity card (RFID) near the device, for it to read.

Utilities

In the Utilities category, there are several generic tools that are reusable on different systems. The currently implemented utilities comprise I/O, layout, encoding and byte/string operations. The application is developed with modularity in mind, so that one specific function, (e.g. the home screen) may easily be exchanged with another.

[6]We do not address how to do this securely or argue on whether such a shared OffPAD is a good idea. After all, it would no longer be a *personal* authentication device.

Authentication service

The authentication service module is a large module, in itself containing other modules. Exchanging digest authentication for another authentication scheme should be fairly simple. One could also change the protection mechanism for stored and transferred credentials (one-way function), which we suggest in chapter 5 may be an option.

Graphical User Interface

The GUI for the TazCard application is developed using Sun's Lightweight User Interface Toolkit `com.sun.lwuit`. We have manually included navigation between screens by finger (swipe gestures) through the J2ME touch screen function `pointerDragged`.

Communication

While the TazCard supports several communication technologies, the only developed module is USB communication. This connection handles all communication to the client computer. It may be exchanged with another communication technology implementation. In the case of USB communication, the TazCard appears to the computer as a Network Interface Card (NIC) accessible through the static IP address `192.168.42.1`. Unencrypted communication is only allowed through UDP port `4242`, but encrypted protocols SSH (secure shell) and SCP (secure copy) are also supported. SCP can be used to copy files between the TazCard's file system and the computer.

Logging

Support for logging is also present in the OffPAD. The ability to audit the actions and interactions on the device is leveraged by a sufficient logging procedure. The TazCard is also able to log to standard output (*stdout*) at runtime, using common Java commands.

4.1.2 Running the Midlet

Here we describe both the registration phase and the authentication phase as use cases, with references to functions in the software.

Registering a new online identity

Before authenticating users to online services, the TazCard reads each identity object[7] into memory from a series of credential files in the `/users/` directory on the TazCard. To register a new credential set on the TazCard one must enter or create a file in the `/users/` directory and append a line containing the credentials. These credential files are formatted like

[7] `no.uio.ifi.tazdigest.identity.Identity`

Apache's access control lists (see figure 2.9). The application must be restarted to be able to recognize and load the new identity object.

Authenticating with a stored online identity

With the browser plugin installed in Google Chrome, every supported address that produces a HTTP Digest Access Authentication challenge will automatically be transferred to the TazCard via USB. When an authentication request appears on the TazCard, the thread `ChallengeWatchThread` will receive it and let `NetworkCommControl.parseAndRespondToChallenge()` look for a realm supporting the challenge. If an identity is found (i.e. the TazCard has a stored `HA1` for that combination of realm and username), `HA1` is loaded into the function `NetworkCommControl.calculateAndPassBack()`. Here a `response` value is calculated using an embedded, freely available MD5 library written by Paavolainen and Macinta [24].

Authenticating with an RFID identity card

If the authentication challenge appears at the TazCard and no corresponding stored identity is found, the device alerts the user and requests him to present an RFID card with the identity object stored on it. When one subsequently appears within the reading range of the device, the same calculation as above is executed, now based on the identity stored on the card. The MIDlet has an always-invoked listener for NFC events, `CommunicationListener`. By the interface `TargetListener` it executes the method `targetDetected()` whenever an RFID card is discovered. When an unknown identity is used on the OffPAD, the user is queried to store it on the device for later use.

4.1.3 Storing identities on NDEF cards

The initial intention with the NDEF RFID cards was to store entire identity triples, ideally encrypted with the device's unique public key and signed by the issuing party. It turned out that the bundled cards' storage is limited to 42 bytes. Therefore, for the convenience of longer usernames and realm names[8], we decided to compress the identity triples using a very computationally light algorithm based on Huffman encoding.

We assume that the average identity triple consists of eight characters for each the username and realm, two colons and 32 characters for the 16 octets of the MD5 hash[9]. These sum up to 50. As we represent the `HA1` value in hexadecimal format, we decide priority for the hexadecimal numbers in the encoding scheme – they are used in at least $\frac{32}{50} = 64$ % of the triple. The rest are the ordinary ASCII letters ($\frac{64}{50} = 32$ %), not to forget the colon (:), always appearing at two spots ($\frac{2}{50} = 4$ %). We use this probability of occurrence as a basis for the total character weight. The non-hexadecimal

[8] As the `HA1` value is fixed-size, only these two values interfere with the total size.

[9] We decided to represent the `HA1` value in hexadecimal following Apache's formatting of ACL files.

Character	Adjustment	Weight	Encoded bits	Char.	Adj. (%)	Weight	Bits
:	0	0,04	0101	t	8,21	0,017	111000
0	0	0,04	0110	o	6,01	0,017	111001
1	0	0,04	0111	i	5,21	0,017	111010
2	0	0,04	00100	n	4,91	0,017	111011
3	0	0,04	00101	s	4,21	0,017	111100
4	0	0,04	00110	h	3,91	0,017	111101
5	0	0,04	00111	r	3,81	0,017	111110
6	0	0,04	10000	l	0,9	0,016	111111
7	0	0,04	10001	u	-1	0,016	110000
8	0	0,04	10010	m	-1,5	0,016	110001
9	0	0,04	10011	w	-1,5	0,016	110010
a	0	0,04	10100	g	-2,1	0,016	110011
b	0	0,04	10101	y	-2,1	0,016	110100
c	0	0,04	10110	p	-2,2	0,016	110101
d	0	0,04	10111	v	-3,5	0,015	110110
e	0	0,04	01000	k	-3,9	0,015	110111
f	0	0,04	01001	j	-4,8	0,015	00000
				x	-4,8	0,015	00001
				q	-4,9	0,015	00010
				z	-4,9	0,015	00011

Table 4.1: The static Huffman encoding scheme

characters (g-z) are additionally weighed relative to each other. We weigh these by each letter's frequency in the English language documented by the Oxford Dictionary [43]. Each of these 20 remaining letters' probability of appearing as one of the 26 English characters is listed as the row "Weight" in table 4.1.

As a result from the weighting get the substitution table marked "Encoded bits". We then use this for compression and decompression for every piece of information to transmit via spare storage media such as NDEF. We decided to use a static substitution table to avoid additional analysis workload on the handheld device.

4.1.4 HTTP Digest Handler for TazPhone

The TazPhone (TPH-ONE) is TazTag's Android mobile phone, and is the first secure mobile phone with NFC and ZigBee communication [39]. Its secure element facilitates secure storage and handling of critical data, such as the `HA1` hash values. The application is written in Java using the Android library, built by Eclipse and the Android Development Tools (ADT) Plugin.

As mentioned, lacking a proper TazPhone testing device, there is no support for communicating with the secure element in this application. The secure element functionality is therefore only shown in the MIDlet (section 4.1). As a result of the minimal development, the TazPhone application is sparsely described in this report because of its similarity to the TazCard application. The functionality of the TazPhone application would most likely be equal to that of the TazCard. From the architecture presented in figure 4.1, the GUI and communication technology modules must be rewritten to support the Android user interface and NFC

communication.

4.2 HTTP Authentication Extension for Google Chrome

Challenge-response authentication using an external device introduces the issue of how the challenge should be transferred to the device. In section 3.7 we argue that the most reasonable and usable solution is having a web browser extension collect the challenge. A browser extension is a great way of providing enhancements to the browsing experience. It is placed at an abstraction layer well suited for slightly altering the user's interaction with the web browser. The extension must be simple in use and installation, and require little or no knowledge and effort from the user; all the inner workings of the extension should be abstracted from the user.

The TazCard browser extension is written in JavaScript as a browser background script (explained below in section 4.2.1). It uses the Google Chrome `webRequest` API for catching authentication headers related to HTTP Digest Access Authentication challenges. This happens in the event `chrome.webRequest.onHeadersReceived`. In the browser window, an icon is visible, indicating the current running state of the extension, and is also used to toggle the state: a light green icon indicates that the extension is enabled and accepts incoming authentication challenges, and a light red icon indicates that the service is disabled.

Whenever the extension notices that the browser has navigated to a supported site, it takes action in the `headersReceivedListener()` event listener. The `WWW-Authenticate` response header is stripped from the response, and its values extracted. Immediately, the `postToTazCard()` function passes an AJAX[10] HTTP `POST` to the TazCard with an appended request header called `OffPAD-Authenticate`. Within are the fields from the `WWW-Authenticate` header, `realm`, `nonce`, `algorithm`, `qop`. The address of the resource, `uri`, is also appended, after `method`. HTTP `GET` is the only supported HTTP method. The `OffPAD-Authenticate` header is quickly received by the OffPAD and handled in its running listener application, desribed above.

4.2.1 Background script

While browsing the Internet, a series of events occur in the web browser, and in the case where a background script is enabled, JavaScript code linked to these events may be executed. Table 4.2 shows a selection of browser events related to header transmission and their JavaScript counterpart available in Google Chrome. The list is adapted from Google Chrome's extension documentation [11]. In figure 4.2 we show an example of how a background script can be used to show an alert whenever a response is about to be sent (the `onResponseStarted` event fires).

[10] Asynchronous JavaScript and XML (AJAX) is a collection of technologies providing asynchronous web requests outside the normal browser context.

Event	JavaScript event	Description
Authentication Required	`onAuthRequired`	Normally shows the username and password dialogue. Used to modify this behaviour.
Request initiated	`onBeforeRequest`	Before TCP connection is set up. Used to modify or stop connection.
	`onBeforeSendHeaders`	Before headers are sent from client. Used to add, remove or modify existing headers.
Response received	`onResponseStarted`	Informs when the server response appears at client, before it is processed.
	`onHeadersReceived`	Before headers are received at the client. Used to modify incoming headers.

Table 4.2: Different browsing events in Google Chrome

```
1 chrome.webRequest.onResponseStarted.addListener(
2         function(details) {
3                 alert('foo');
4         }, {urls: ["<all_urls>"]},
5         ["responseHeaders"]
6 );
```

Figure 4.2: A background script linked to a browser event

In this script, whenever a response is to be created, the message *foo* is shown.

4.3 Technical execution flow

In the following we present each step of the user authentication process in detail. Each numbered point from figure 4.3 is explained, in a scenario where a user navigates to a protected remote realm to which he must authenticate for access.

1. The HTTP Authentication Extension for Google Chrome has been enabled in the client web browser to act on incoming headers. The initialization function `init()` in the background script (`background.js`) adjusts the browser to call the function `headersReceivedListener()` upon every `chrome.webRequest.onHeadersReceived` event. The client web browser navigates to `http://heim.ifi.uio.no/hennikl/protected/` to retrieve the root document.

2. The server responds with a `401 Authorization Required` HTTP status code containing a challenge to the client.

3. (a) Before the challenge has been interpreted by the web browser, it is caught by the `headersReceivedListener()`

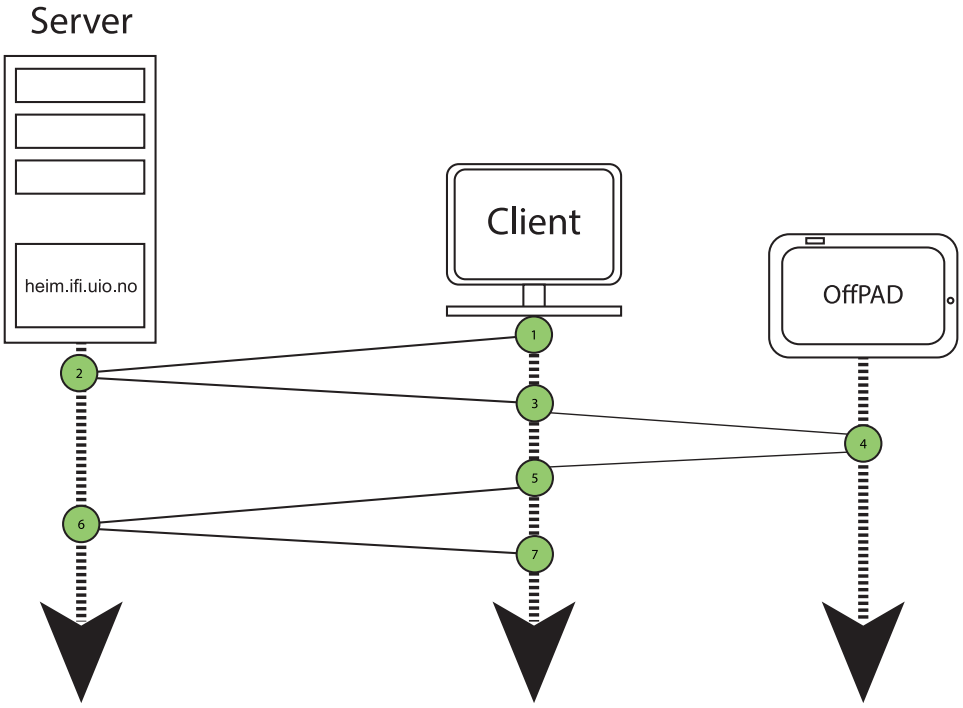


Figure 4.3: Technical execution flow (server perspective)

listener function, which validates `http://heim.ifi.uio.no/` as a supported URL (using the function `isSupported()`).

(b) The listener iterates the response headers (`details.responseHeaders`) and locates the `WWW-Authenticate` header.

(c) The function `extractValues()`, extracts the challenge data: `realm`, `nonce`, `algorithm`, `qop`, `method` and `uri`, and returns a string with these concatenated.

(d) Next, the `postToTazCard()` function builds an HTTP request to the TazCard. Here, the challenge information is converted into hexadecimal, and entered into a request header called `OffPAD-Authenticate` for the OffPAD to interpret.

(e) The TazCard is assumed to be connected to the computer at IP `http://192.168.42.1:4243`. The request built in `postToTazCard()` is sent as an HTTP POST request, and the execution of the background script temporarily ends.

4. (a) At the TazCard, the thread `ChallengeWatchThread` is running, and is activated when the HTTP POST request arrives from the computer. In it, the simple function `NetworkCommControl.cleanFromComputer()` is invoked, to strip the incoming request of the `OffPAD-Authenticate` header. The header is converted into ASCII and the challenge values are returned as a byte array.

(b) The challenge is passed to the function `NetworkCommControl.parseAndRespondToChallenge()`, where it is converted to

an `AuthenticateChallenge` object. When the TazCard application was started, all possible identities were loaded into the application's memory, located in `IdentityManager.identities`. These are retrieved by the function `IdentityManager.realmHasALoggedInIdentity()`.

(c) If there are several identities supported for the requested realm, an identity selection screen is invoked, from which the user must select what identity to assume. If there is one, this identity is used. If no identities are available, the user is asked to present an RFID card with a valid identity. If unable to do so, execution ends.

(d) When the correct identity is selected, `NetworkCommControl.calculateAndPassBack()` is called, with the challenge and the selected identity as parameters. The class `ResponseCalculator`, which takes care of HTTP Digest Access Authentication, calculates the `response` value from the parameters, and returns the resulting bytes.

(e) A set of response parameters is built, consisting of values from both the identity, challenge and the calculated `response`. A static `cnonce` is used in the implementation. The parameters are returned as an `InfoPiece` object to `ChallengeWatchThread`. In total, it comprises: `username`, `realm`, `nonce`, `uri`, `algorithm`, `response`, `qop` and `cnonce`.

(f) A HTTP 200 OK is written, as a response to the POST issued from the browser. Appended to it is the aforementioned parameters, in an `OffPAD-Info` header.

(g) The `ChallengeWatchThread` returns and starts listening for more challenges.

5. (a) When the TazCard has responded to the challenge the browser background script can return and work on the received response.

   (b) The function `authenticateTo()` takes the header collected from the TazCard and uses it to build an `Authorization` header to use against the remote server. The parameters received are split and placed into their respective location in the `Authorization` header. The resulting header is equal to what would have been generated by the browser in the original case, with the user entering his credentials manually. The Authorization header is sent to the remote server from the browser extension, whose execution terminates until the next challenge appears.

6. The server receives the Authorization header from the client's browser. As there is no difference between the original digest scheme and the way the TazCard produces its `response`, the server's calculation of the `response` value is equal. If the user

credentials (the `HA1` value and the user name) on the TazCard are correct, and the user is authorized access to the realm, the server approves access.

7. The client receives a HTTP 200 OK status from the server, along with the contents of the protected resource.

Chapter 5

Security testing

Below we present different experiments on trying to interfere with, or compromise the authentication process we have described above. We comment on the feasibility on each of the approaches and how, based on the results, the process should be strengthened.

As in all the implementation examples, we also here assume Apache httpd is running as the server side application. This means that all experiments have been done using httpd and that other server applications may react differently.

The experiments show different attack vectors in the entire system described in this report. When attempting to abuse the system, there are many different angles to choose from. The motivation for attacking an authentication system may be to harvest user credentials, gain unauthorized access to the systems to which the user authenticates, or both. Also, the system may be exploited to carry out unauthorized authentication. If the device, the application or the communication process is not protected sufficiently, the user may fall victim to identity theft.

5.1 A comment on compliance with the TOCTOU principle

By looking into `Authorization` headers passed to the server on several `HTTP GET` requests over time, we can confirm that httpd's standard for nonce lifetime is set to 300 seconds[7]. When the nonce lifetime has run out, the client is initially unaware. It will try using the old nonce but is notified by the server after the first failed attempt. When an old nonce appears at a server, it responds with a new `401 Authorization Required`, a fresh nonce value and the `stale`-flag set to `true`, indicating to the client that the nonce was aged and has been replaced. This method is specified in the RFC[9]. If the user agent has the credentials remembered, the transition to a fresh nonce will be done automatically and a valid `response` value is calculated, re-authenticating the user. However, the TOCTOU principle (which we introduced in section 2.1.2)

does *not* hold, as this passive re-authentication is not another time of check. It does not update the system's assurance that the identity is correct. For the $\Delta TOCTOU$ (i.e. the time between check and use of the credentials) to be as short as possible, the user should be forced to physically re-enter his credentials to reassure the system. With an OffPAD, this is equivalent to a physical re-authenticating by initiating communication between the device and the computer.

5.2 Attacking the challenge-response protocol

From figure 2.10 we can see that the realm- and username is used as a *salt* to the hash function producing `HA1`. As the password is not hashed alone, one cannot retrieve the password through a (pre-calculated) rainbow table. A rainbow table for the purpose of attacking a password within a `HA1` value must be pre-calculated for one particular combination from the set of realms and usernames. In the `HA1` calculation (section 2.5.4), the static values (username, realm and colons) are analogous to *salt*. In the `response` calculation (section 2.5.4) we consider the `HA1` value analogue to *password* and the static values (nonce, nc, cnonce, qop and HA2, all separated by colons) analogue to *salt*.

We have the following:

1. One or more `HTTP GET` requests to a resource containing an `Authorization` header are *snooped*, i.e. read off the network cable or wireless channel by a *Man In The Middle*.

2. A possibly high number of different `response` values. These are hashes of the same combination of header data and a different, known salt (`nc`) each time.

An `Authorization` header's `response` value is an expression on the form:

$$HA1 = MD5(s_1||password)$$
$$response = MD5(HA1||s_2)$$

Where $||$ describes string concatenation, s_1 and s_2 are the static values, of the format "username:realm:" and "nonce:nc:cnonce:qop:HA2" respectively.

A successful brute force attack on the password will reveal the static secret *HA1* value that in turn can be validated by the *response* calculation above.

As stated in section 2.5.5, attempting to break the one-way property of MD5 is not practical at the time of writing. The most effective known preimage attack has a computational complexity of $2^{123.4}$[33]. To find a usable password[1], however, we must find the preimage of the `response` value, which itself is hashed.

[1] Although this is a fully usable password within the system, it is only so because the calculations yield the same `response` value. There is no certainty as to whether the password retrieved is the password the user originally selected, so it is not guaranteed that a successful brute force attack yields a password that is transferable to other systems.

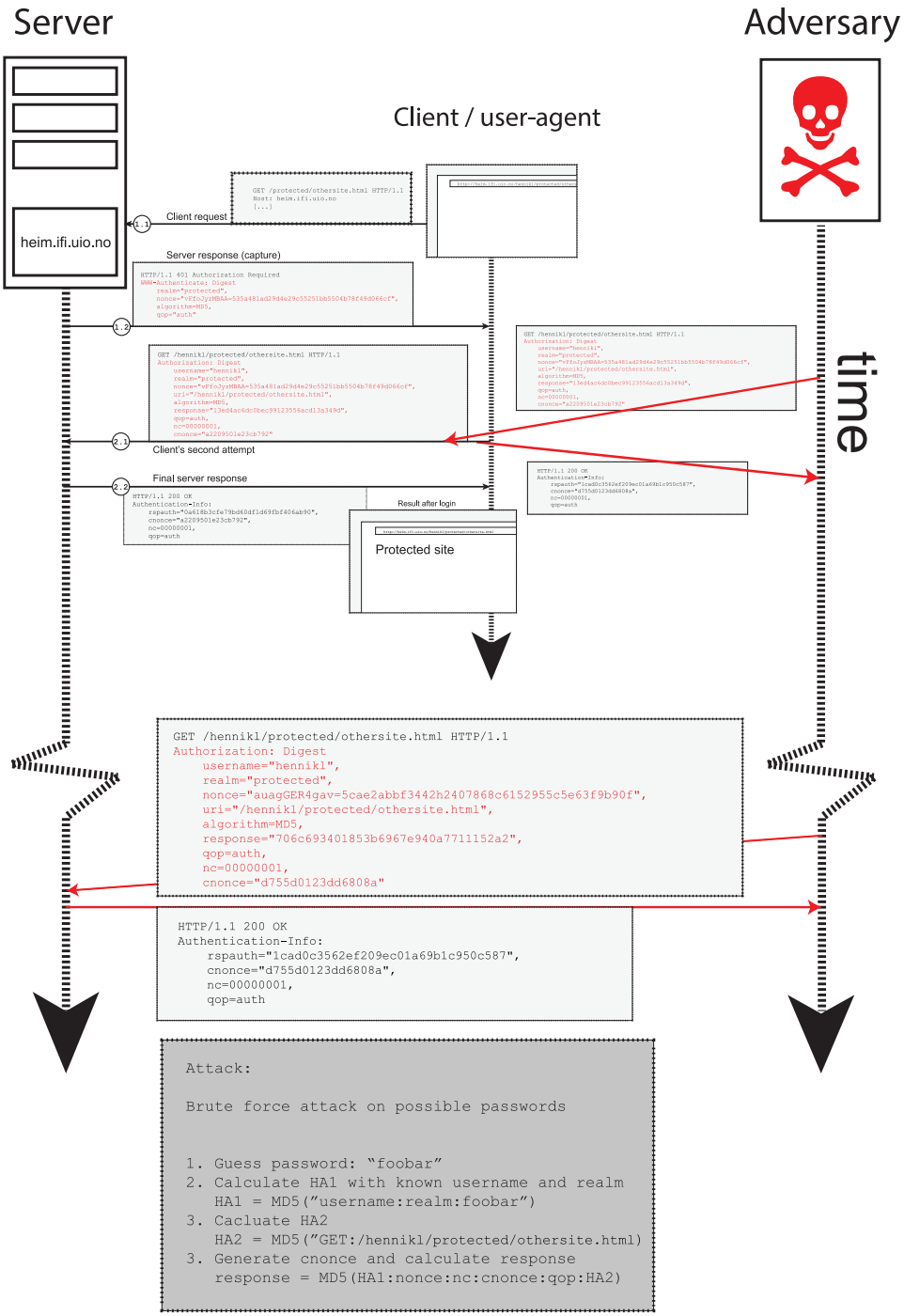


Figure 5.1: An attack on digest authentication

password	HA1	response
hennikl:realm:a	7a29f74992 ... a0fdda	8f2a65080a8761d4ee6da59544ccc186
hennikl:realm:b	e3e5f66d00 ... de36f6	61f33b9a185989d9f215c916e67afa68
hennikl:realm:c	53d83d97af ... 05404e	10bb7eca0fcf8876b3cf8a44fa98a185
hennikl:realm:d	9ef81d5334 ... 769b8a	beaa89ecc4c7f7863982ff1f65efd65e
hennikl:realm:e	a0b1fae479 ... 397e1d	d17b3e2a0ddc857329c3aa4df232736d
hennikl:realm:f	5a991fb4a4 ... d5fc0c	49f275d192e250e8a5787284298f8e05
hennikl:realm:g	4f71d53117 ... 84d22d	3cc9c20d9a0006cb8cc371b3c873aec4
hennikl:realm:h	dabeeba885 ... 78a195	d3c91f4d2fe3ca8ba28f9bb3cbefd4b9
hennikl:realm:i	115dd27a08 ... b633ca	1183e7df1779e43473e26e0febd6148d
...		
hennikl:realm:passwor5	9684b080fa ... 9901f4	b815adac2c1045b40ed621c347b661a8
hennikl:realm:passwor6	3caa6da3aa ... 69b735	31380b94e595f449ccafc2bf9063ff2b
hennikl:realm:passwor7	14fe235f79 ... abb63c	35127fbc6b025ccc8540b690d7958013
hennikl:realm:passwor8	b1a7fe3369 ... f0966c	26794bc1025236342aee2c653d52a7c9
hennikl:realm:passwor9	7f8f6ef704 ... 7c0ff3	c7b18e356f24ed141dbd9e0cf3722a89
hennikl:realm:passwora	d356381226 ... 9cd4e3	28aabbb0f847fdabc899ae0f949caae7
hennikl:realm:passworb	5b4415f182 ... 7b8b95	9a816243210af42e50767f61ba0fea7c
hennikl:realm:passworc	e281050e8f ... 50f93a	d62ef054edaa5b216db454e565e36a0a
hennikl:realm:password	f19b0a03ee ... 471dcf	**f15370722fb0a84b799c669cdb4b35d6**

Table 5.1: An example of an exhaustive search

However, attacking from another angle is possible. At the same time as we collected the `Authorization` header, we also collected all the values needed to calculate the `HA1` except the password. Actually, all values making up the entire final `response` are accessible should we find the correct password. Exhaustively searching the available preimages' character space is a usual approach to *password cracking* on hashed passwords. In this scenario we must customize the password cracking algorithm to first hash the guessed password together with the rest of the `A1` parameters to recreate a suggestion for `HA1`. Second, we must use that `HA1` value in the `response` calculation (using the retrieved nonces and other collected parameters) to produce a possible response value. Finally, in the targeted event that the `response` value equals the one collected, we have guessed a password that is usable at least in any session with the same system. We have decoupled the password from the nonce- and client nonces. In table 5.1 and figure 5.1, we show how this approach is possible, using these example values:

```
username: hennikl
realm:    realm
nonce:    aGVsbG8=feffda0520707fc331d9be9eff74eab1eb7cafe4
cnonce:   SHZlbSBoYWRkZSB0cm9kZCBhdCBkZXQgc3RvZCBub2UgaGVyPw==
nc:       00000001
uri:      /foo/
method:   GET
qop:      auth

correct password: password
correct HA1:      f19b0a03eead15d687986227dc471dcf
                  MD5("hennikl:realm:password");
```

```
correct HA2:      2e18ba280b7f2a4e2785f9d88fc7aa72
                  MD5("GET:/foo/");
correct response: f15370722fb0a84b799c669cdb4b35d6
                  MD5(HA1:nonce:nc:cnonce:qop:HA2);
```

Each of the text values are to be interpreted as strings of bytes. The presented hashes are strings of bytes in hexadecimal (but ASCII) representation. The only value changing between each `HTTP GET` within one such *nonce lifetime* (five minutes) is the nonce counter, `nc`, which is incremented on every client request. Thus is the only value responsible for the change of the `response` value between requests to the same resource. If multiple `Authorization` headers are collected, each have different `nc` values, but may be attacked in parallel processes, each having equal probability of recovering the password.

5.2.1 Pseudocode

This section contains pseudocode showing the execution of the exhaustive attack on the digest authentication scheme. It follows the same general recipe as any brute force hash cracking algorithms, namely going through an entire dictionary until the attack succeeds. Algorithm 1 shows the calculation of the `response` value. Algorithm 2 iterates through a stack of suspected passwords in a dictionary, passing each password suggestion to the response calculator. It requires the data from one intercepted `Authorization` header. The dictionary can of course contain anything from arbitrary bytes to English dictionary words.

The procedures `DigestCalcHA1`, `DigestCalcHA2` and `DigestCalc Response` are equal to those specified in RFC2617[9]. The one-way function, however, which is `MD5` in the specification, may be regarded as any other one-way function as long as it is used equally on both endpoints of the authentication. Synchronization of the selected one-way function is done using the `algorithm` field.

Procedure: CalculateResponse
Data: pszMethod, pszDigestUri, pszQop,
pszUsername, pszRealm, pszPassword,
pszNonce,pszCNonce, pszNonceCount
Result: pszResponse
$ha1 \leftarrow DigestCalcHA1(pszUsername, pszRealm, pszPassword)$
$ha2 \leftarrow DigestCalcHA2(pszMethod, pszDigestUri, pszQop)$
$response \leftarrow DigestCalcResponse(ha1, ha2, pszMethod, pszDigestUri,$
$pszQop, pszNonce, pszNonceCount, pszCNonce)$
return *response*

Algorithm 1: This procedure calculates the `response` value

```
Procedure: DigestDictionaryAttack
Data: pszMethod, pszDigestUri, pszQop,
      pszUsername, pszRealm, pszNonce,
      pszCNonce, pszNonceCount, pszTargetResponse
Result: pszPassword or null
while dictionary is not empty do
    pszPassword ← dictionary.pop()
    res ← CalculateResponse(pszMethod, pszDigestUri,
          pszQop, pszUsername, pszRealm, pszPassword,
          pszNonce, pszCNonce, pszNonceCount)
    if res == pszTargetResponse then
        return pszPassword
    end
end
return null
```

Algorithm 2: An approach to retrieving a password used in HTTP Digest Access Authentication

5.3 A phishing attack on the OffPAD

Assuming a trojan on the client, the address transferred to the OffPAD cannot be trusted. Navigating to a phishing site, e.g. deployed in Ukraine at `http://paypa1.ua/`, the Ukrainian address may arrive at the OffPAD changed to the correct domain address `https://paypal.com/`.

The change may happen in (or en route from) the browser extension handling the incoming authentication headers, or the phishing site may actually embed the correct PayPal login screen on the phishing site, invoking the OffPAD's authentication routine for that server. Both scenarios release a `response` value to the phishing site, which if not protected well enough, may be vulnerable to exhaustive search attacks. This is another argument for security controls beyond the OffPAD device itself. If the password is properly protected, e.g. by key derivation, which we discussed in sections 2.3.1 and 5.2, the complexity of recovering the password from the `response` value quickly becomes too large to handle.

5.4 Local collection of password

In the original HTTP Digest Authentication scheme, running on the user's computer, credentials are entered in the browser by keyboard. As the keyboard signals travel from the keyboard to the application, they are susceptible to surveillance on many layers (see figure 5.2):

1. The lowest-layer way to do keylogging is by collecting the signals sent through the USB connector. This can be done by connecting a hardware keylogging device between the USB socket of the computer and the connector of the keyboard. Hardware keyloggers

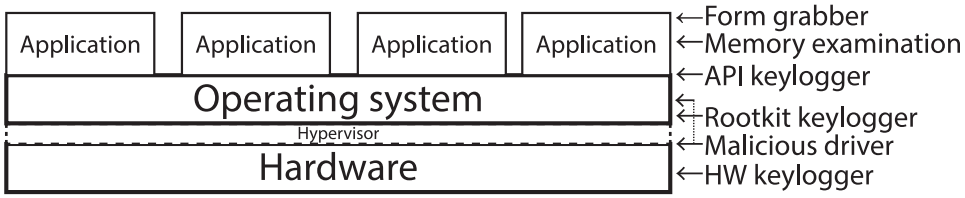


Figure 5.2: Locations where the password is vulnerable

store whatever keys are pressed in its own local storage. The keylogger device must be collected later, and the data extracted from the device. Being physical devices, hardware keyloggers have a high risk of discovery.

2. Malware installed as an interim layer (hypervisor) between the hardware and operating system has direct access to hardware, as well as full control over how the operating system uses it. With these abilities the hypervisor is able to on its own define how the system should react to keyboard signals. The trojan can store the typed keys and transfer them without alerting the operating system.

3. Malicious drivers may also be installed in the operating system layer, either by unwitting consent of an administrator or by a falsely trusted and authenticated driver certificate.

4. A *rootkit* is malicious code that has gained *root-* (kernel level) access, thus enabling access to hardware resources, including the keyboard. Rootkits have the ability to hide while passively collecting keystrokes.

5. API keyloggers exploit methods in the operating system for listening in on keyboard signals. These usually introduce significant overhead to the system, which may alert the user.

6. Every piece of information that is written on the keyboard and presented in the system is loaded into memory. From here, one could apply several *data mining* techniques for collecting credentials information. One would look for passwords in the memory context of a web browser login, and credit card numbers can be validated by Luhn's algorithm. In figure 2.4 we show how a password entered during original digest authentication is exposed in memory. Because of this, it is theoretically possible for an attacker to isolate and collect the password at a compromised machine.

7. Form grabbers are software that steals information from the browser context. When entering information in a web form (e.g. text fields for entering username and password), the content of the form can be collected by intervening malware. An example

implementation of a form grabber can be a benign looking browser extension. Having access to the web page (the DOM), the trojan may easily collect form data by walking through the appropriate text fields.

5.5 Forced backwards compatibility (MITM)

Where most systems' backwards compatibility with earlier versions can be helpful for usability and error transparency, it has no place in an authentication scheme. If the server decides (or is forced) not to set the quality of protection field, both endpoints will roll back to RFC2069[8] and be subjected to its weaknesses without the user knowing. As we mentioned in section 2.5.7, our solution avoids this by not supporting backwards compatibility.

5.6 Security architecture of the OffPAD

This is an introduction to the security architecture of the OffPAD. Below we assess the possible threats presented in table 5.2, and comment on what protective services can be implemented. Some, but not all of the security services are implemented in the TazCard application.

#	Threat	Security services
1	Unauthorized disclosure of identities	Device access control
2	Unauthorized use of device	Authentication on device
3	Unauthorized use of unlocked device	OffPAD session timeout
4	Creating false identities	Phishing protection / server authentication
5	Spoofing authentication	Server authentication & protection of `response` value

Table 5.2: Attack scenarios and protection mechanisms on the OffPAD

5.6.1 Attack scenarios

These scenarios apply to the TazCard OffPAD specifically.

1 - Unauthorized disclosure of identities

The original configuration of the TazCard provides low protection of the file system. Given the device password, which is easily guessed, the entire file system is readable via SSH. The authentication application works with identities that are either

(a) persistent, and stored either on-board (`/data/midp/jstore/`) or on a MicroSD card(`/sdcard/tazcard/`).

(b) transient, created by the user before they are required, or upon request. Such identities can be read from and written to supported NDEF RFID ("contactless") cards (see section 4.1.3).

In both cases, the stored identities (either on RFID or MicroSD card) are stored in the ACL format from figure 2.9, which reveals the `HA1` value. If a `HA1` value is revealed, it can be used to calculate a working `response` in the same way as the OffPAD.

A possible measure against this, which is not implemented, is to encrypt the identities while stored. This could be done using a key only accessible by the intended owner, perhaps derived from the device master password, using a key derivation function. This attack scenario is plausible, as getting hold of the identities is the most valuable objective.

2 - Unauthorized use of application

If a TazCard is stolen or used by an unauthorized person, there might be attempts to perform authentication on the services that the device has a relationship with. To counter this, the device's user must be authenticated to unlock an *authentication session* which in turn unlocks the functionality of the TazCard, the user's profile and access to his identities.

3 - Unauthorized use of unlocked application

In the scenario above, if the device is unlocked by an authorized user at the time of use, it can be abused to access all protection realms to which the owner is authorized. The impact level of this abuse can be reduced by introducing a session timeout on the device. This way, an attacker has limited time to commit unauthorized actions. While he is also able to modify and delete identities on the device, there is no direct way for the attacker to recover the passwords on the device.

4 - Creating false identities

While authenticated to the device, an attacker may create or more "false identities" on the device. Such an identity may be used to authenticate to a phishing site, seemingly equivalent to another, to which the authenticated user already has a relationship. The user may be lead into believing that the web site to which he is automatically authenticated is the correct one. Other phishing vulnerabilities are addressed by Kent Varmedal in [41].

5 - Spoofing authentication

An attacker could either try to intercept and relay (MITM) real authentication requests, or create false ones from a collected `HA1`

or password. However vulnerable to interception, the information communicated from the OffPAD to the computer is transient and usable only once, between the correct endpoints and during a specified time interval. Strong cryptographic protection of the transmitted `response` value provides better protection of the contents of the `response` value and thus its contents (`HA1` and the password). For the man-in-the-middle situation, server authentication is needed to avoid the client releasing valid `response` values to malicious "middle men".

Chapter 6

Conclusions and future work

6.1 Conclusions

In this report we have presented a method and device (the OffPAD) for moving the operations of the authentication phase external to the computer. This benefits the user in the following ways:

1. Management of credentials and calculations using these credentials is done external to the computer. This means that clear text passwords will never appear on the client computer and never be revealed in its memory. Credentials are rather stored on the device in the `HA1` format. As an addition to the original MD5 hashing of `A1`, we propose that the value is better protected in storage if either the password is set to a random value, or protected by an iterative or workload-adjusted key derivation function. These mechanisms mitigate the vulnerability of password disclosure while stored.

2. Calculation of the critical values is done external to the computer. This limits the availability to the critical values (the password and `HA1`). The password itself is no longer a direct part of the challenge-response protocol, and the `HA1` value never leaves the OffPAD. Doing the (passwordless) extended digest calculation on the computer directly, one would never expose the password in clear. However, as the `HA1` value is still used in the calculation, it must at some point be present in memory. As explained, `HA1` is a sensitive piece of credential data and may be replayed.

3. Transmission of the critical values is secured. We have shown that today's digest authentication scheme is insufficient in its protection of the `HA1` value, and presented in chapter 5 several ideas for securing the communication of the scheme. We do not require or assume any additional security on the communications link, such as SSL/TLS or encrypted IP (IPSec).

4. User interaction is reduced, both in time and effort. The user does no longer have to remember passwords for several accounts. There is only one master password to manage, and all passwords

used in the managed accounts' credentials now reside in the `HA1` value stored on the device. The credentials are applicable for low assurance level accounts as well as high assurance accounts. Banks or governments may issue their electronic identities on protected smart cards for the user to store his proof of identity on the device.

In the security testing phase of the thesis work, we uncovered several weaknesses, and pointed to where the original digest authentication scheme requires modernization, i.e. where its protection mechanisms are lacking. This led us to suggest improvements on how user credentials are handled on the device, and the extension of the digest authentication scheme, where we showed how user authentication can be done more securely. As can be read in sections 6.2 and 6.2.7, there is still a lot that remains to be done.

The OffPAD solution may be used as a standalone authenticator, or alongside other mechanisms to complement a multi-factor authentication scheme. The technology behind the scheme is not particularly advanced, and does not differ greatly from its original HTTP Digest Access Authentication scheme. This fact may leverage adoption of the scheme by web services.

6.1.1 Research questions

Here we briefly recapitulate on the research questions raised in section 1.2.

1. What opportunities does the OffPAD offer?
   The OffPAD provides a novel approach to automatic user authentication and identity management. The user's local management of his own credentials is done on the external device (OffPAD) rather than in the user's brain. This provides users with a more effective, less tiresome identity management, and quicker authentication. The weakening requirement of having a password that is easy to remember is now removed, as passwords can be arbitrarily long. Thus the solution also adds to the security of the scheme.

2. What has already been done in the field of local user-centric identity management?
   Several devices similar to the OffPAD exist, both in literature and research. These are presented and discussed in sections 1.5 and 3.2.

3. What security requirements must be considered for a contemporary user authentication scheme?
   Extensive effort was put into describing the security requirements for our system and to suggest improvements for the original HTTP Digest Access Authentication scheme. We particularly considered communication security, where the original scheme was lacking.

Chapter 5 presents the requirements, issues and proposed solutions to the scheme's security related shortcomings.

4. What is the state of HTTP Digest Access Authentication?
   Section 2.5.7 was written to address this question in particular. Here we present the issue of brute force vulnerability and argue why the scheme is still usable, with only minor modifications to the password. Chapter 5 goes into detail on exploiting the findings in section 2.5.7.

6.2 Future work

The research topics dealt with in this thesis (security; usability; identity management; access control; etc.) are all popular research topics, and are highly relevant to solving contemporary user authentication security and usability issues. The concern of the user is often overlooked by service providers in the planning and implementation phases of identity management systems. The following is a presentation of ideas for improvement, and new applications of the OffPAD, software and research topics described in this report.

The security solution described in this report is applicable in many situations. The original intention has been using the OffPAD as a user-centric identity manager and authenticator, but the software can easily be reduced or improved in many ways. Using the OffPAD's secure element, we can provide secure management and storage of data that require protection services such as confidentiality and integrity, as well as using the device as an external, hardened service for doing cryptography and digital signatures.

New applications may require a stronger access control to the device than what has been proposed in this report. We have not considered any authentication of the local communicating entities in the scheme, i.e. mutual authentication between the OffPAD and computer. In such a scheme, the OffPAD may be registered to a *trusted* computer, perhaps along the lines of the *pairing* suggested for the Pico by Stajano in [35]. The OffPAD could keep a list of authorized computers, and the computer may contain a similar list. For each interaction, or once per session, the entities could authenticate to each other before the exchange of credentials.

The OffPAD's secure element facilitates operation in various fields. It is capable of running cryptographic functions for information confidentiality as well as storing symmetric and asymmetric cryptographic keys as easily and secure as user credentials. Below are a number of suggested new applications for the OffPAD, and finally a recommendation for further development of the applications.

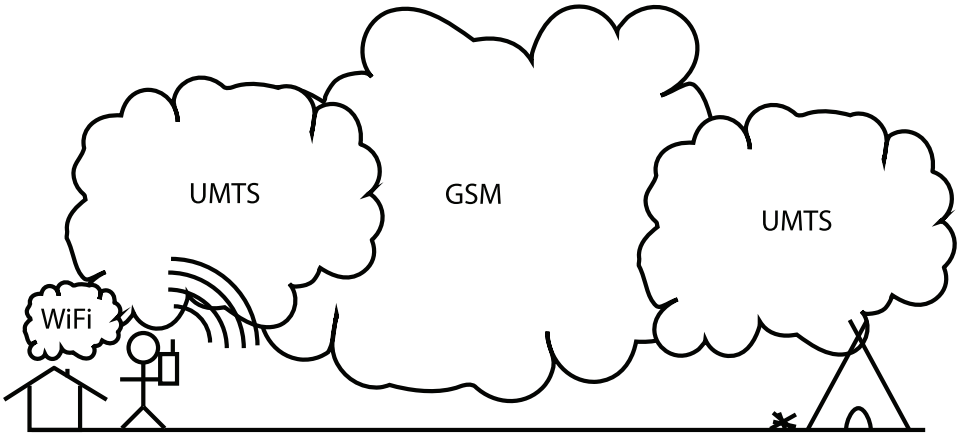


Figure 6.1: Using the OffPAD to facilitate seamless authenticated inter-media handover

6.2.1 Automatic authentication on wireless AP handover

When a mobile user, either represented by Mobile Equipment (ME) or a Mobile Station (STA) moves out of the range of an access point (AP) or base station (BSC) but is still covered by another form of communication technology to which that user is authorized access, the ME or STA may automatically be approved access using the OffPAD as a credential manager and authenticator for wireless media (see figure 6.1). In theory, the automatic authentication mechanism is applicable to any wireless technology (e.g. WLAN, WRAN and GSM). In particular, this is an interesting application to the IEEE 802.21 standard [34] for media independent handover in LANs and WANs.

6.2.2 Credit card payment

Online credit card payment ordinarily happens on vendors' web sites, or via a specialized payment service hosted by the card issuer. In both cases the remote service collects the customer's credit card information from input done by the customer, to prove his possession of the credit card. Many vendors also store this credit card information on their systems for later use, and the convenience of their customers. We believe that this storage and transfer of credit card details, however well protected is unnecessary.

Using the same principles as in the presented extension of HTTP Digest Access Authentication, a hash of the credit card details may be used as the static value (`HA1`) in proving possession of a credit card. This scheme would work particularly well in the situations where the online transaction is done directly between the vendor and purchaser, as the credit card information is already established at both locations.

For online payment facilitated by the card issuer, consider the following scenario (with steps emphasized in figure 6.2):

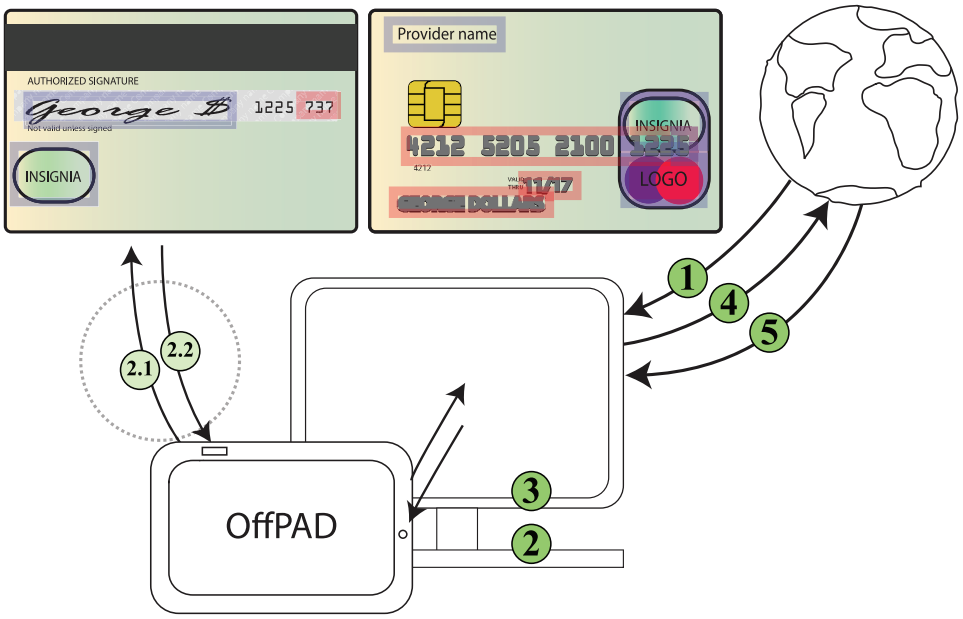


Figure 6.2: Challenge-response verification of credit card details

1. Upon receiving a payment request, the card issuer challenges the client with a nonce.

2. The challenge is intercepted in the browser and passed to the OffPAD. In the device, a digest calculation takes place, using hashed static credit card information (outlined in the figure with red (typed information) and blue (optically recognizable data) frames) stored on the device, as the `HA1` value. If there is no stored `HA1` value for the requesting credit card vendor (consider this the `realm`), the data is transferred from the card. There are several different ways to collect the information in steps 2.1 and 2.2, some including physical changes to the OffPAD device, perhaps a camera or card reader.

3. The `response` value calculated in step 2 is transferred to the computer.

4. Still in the browser context, the computer forwards the received `response` to the remote server.

5. The server validates the received challenge by doing the same calculation locally with stored, correct credit card data. If `response` corresponds to the result of the calculation, the payment is considered authorized by the user and will be carried out.

6.2.3 Online banking

When logged in at an online bank, customers are often prompted to re-authenticate themselves for the act of authorizing payments, signing

contracts and agreements, etc (cf. the TOCTOU principle). As noted in section 2.1.3, the authenticator software on the OffPAD may be used for continuous (stateless) authentication to a web service. This may be applied to the above scenario, as a less interfering substitute to such re-authentication. The act of authenticating with the OffPAD (the physical interaction) may be used as a means of re-authenticating the user without the effort of entering a password.

6.2.4 Encrypted communication

Implementing encryption mechanisms such as RSA or AES on an OffPAD is fairly easy. With implementations of asymmetric cryptography, the OffPAD can sign e-mails and documents, and perform encryption and decryption of data or point to point communication. Plaintext may even be entered on the device itself, so that it is not revealed in the memory of the connected computer.

6.2.5 Storage of cryptographic keys

The OffPAD may be used as storage for X.509 server certificates for SSL/TLS communication, public keys for DNSSEC authentication operations, or keys for encrypted applications such as SSH. Implementing an equivalent of the UNIX application *dig* (domain information groper) combined with an authoritative DNS root public key, we can use the OffPAD as an external, tamperproof validator of DNS data. We can also use the device to validate digital signatures.

6.2.6 Local identity provider

OpenID or similar authentication frameworks may be implemented on the OffPAD, to use as a local identity provider. Such an OffPAD user identity may be used to authenticate the user to any service that support OpenID.

6.2.7 Continuance of the thesis work

The identity management and authentication schemes we have developed and presented in this report are far from perfected. As the TazCard development was suspended in favour of Android late in the thesis work, a great deal of testing and adapting is required for a fully functional OffPAD application for the TazPhone. Usability testing will make it clear whether the proposed solution is desirably developed for end users, and may uncover shortcomings, even with regard to security, that have not yet been considered.

Bibliography

[1] Anne Adams and Martina Angela Sasse. "Users are not the enemy". In: *Commun. ACM* 42.12 (Dec. 1999), pp. 40–46. ISSN: 0001-0782. DOI: 10.1145/322796.322806. URL: http://doi.acm.org/10.1145/322796.322806.

[2] Bander AlFayyadh et al. "Improving Usability of Password Management with Standardized Password Policies". In: *Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI)*. Ed. by Christophe Rosenberger and Mohamed Achemlal. May 2012, pp. 38–45. ISBN: 978-2-9542630-0-7.

[3] J. Arkko and P. Nikander. "Weak authentication: How to authenticate unknown principals without trusted parties". In: *Security Protocols*. Springer. 2004, pp. 57–66.

[4] T. Berners-Lee, R. Fielding and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). Internet Engineering Task Force, May 1996. URL: http://www.ietf.org/rfc/rfc1945.txt.

[5] J. Bonneau et al. "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes". In: *2012 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2012, pp. 553–567.

[6] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266. Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[7] "The Apache Software Foundation". *Apache Module mod_auth_digest*. URL: http://httpd.apache.org/docs/2.2/mod/mod_auth_digest.html (visited on 13/08/2012).

[8] J. Franks et al. *An Extension to HTTP : Digest Access Authentication*. RFC 2069 (Proposed Standard). Obsoleted by RFC 2617. Internet Engineering Task Force, Jan. 1997. URL: http://www.ietf.org/rfc/rfc2069.txt.

[9] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617 (Draft Standard). Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2617.txt.

[10] Dieter Gollmann. *Computer Security (3. ed.)* Wiley, 2011. ISBN: 978-0-470-74115-3.

[11] Google. *Web Requests*. 2011. URL: http://code.google.com/chrome/extensions/webRequest.html (visited on 23/12/2011).

[12] Jeremi Gosney. *Password Cracking HPC*. Rump session, Passwords 12, Dec. 2012. URL: http://passwords12.at.ifi.uio.no/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf (visited on 17/12/2012).

[13] David Gourley and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly & Associates, Inc., 2002. ISBN: 978-1-565-92509-0.

[14] Kirsi Helkala et al. "Cracking Associative Passwords". In: *Secure IT Systems*. Ed. by Audun Jøsang and Bengt Carlsson. Vol. 7617. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 153–168. ISBN: 978-3-642-34209-7. DOI: 10.1007/978-3-642-34210-3_11. URL: http://dx.doi.org/10.1007/978-3-642-34210-3_11.

[15] Audun Jøsang. *Identity Management*. URL: http://folk.uio.no/josang/im/ (visited on 03/04/2012).

[16] Audun Jøsang. "Trust and Identity Management for Internet and Mobile Computing". (Submitted to) IET Information Security - Special Issue on Trust and Identity Management in Mobile and Internet Computing and Communications. 2013.

[17] Audun Jøsang and Simon Pope. "User Centric Identity Management". In: *AusCERT Conference 2005*.

[18] Audun Jøsang, Muhammed Al Zomai and Suriadi Suriadi. "Usability and privacy in identity management architectures". In: *Proceedings of the fifth Australasian symposium on ACSW frontiers - Volume 68*. ACSW '07. Ballarat, Australia: Australian Computer Society, Inc., 2007, pp. 143–152. ISBN: 1-920-68285-X. URL: http://dl.acm.org/citation.cfm?id=1274531.1274548.

[19] Inc. Juniper Networks. *Juniper Mobile Threat Report 2011*. Tech. rep. Juniper Networks, Inc., 2011.

[20] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898 (Informational). Internet Engineering Task Force, Sept. 2000. URL: http://www.ietf.org/rfc/rfc2898.txt.

[21] Henning Klevjer, Kent Are Varmedal and Audun Jøsang. "Extended HTTP Digest Access Authentication". In: *Proceedings of the 3rd IFIP WG 11.6 Working Conference on Policies & Research in Identity Management*. IFIP Springer. London, United Kingdom: Springer, Apr. 2013.

[22] LastPass. *Password Iterations (PBKDF2)*. URL: http://helpdesk.lastpass.com/security-options/password-iterations-pbkdf2/ (visited on 12/12/2012).

[23] B. Laurie and A. Singer. "Choose the red pill and the blue pill: a position paper". In: *Proceedings of the 2008 workshop on New security paradigms*. ACM. 2009, pp. 127–133.

[24] Timothy W. Macinta. *Fast MD5 Implementation in Java*. URL: http://www.twmacinta.com/myjava/fast_md5.php (visited on 17/01/2013).

[25] LLC Mandylion Research Labs. *Tokens*. 2006. URL: http://mandylionlabs.com/products/token.htm (visited on 14/04/2012).

[26] Mohammad Mannan and Paul C. van Oorschot. "Leveraging personal devices for stronger password authentication from untrusted computers". In: *Journal of Computer Security* 19.4 (2011), pp. 703–750.

[27] Alfred J. Menezes, Scott A. Vanstone and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 978-0-849-38523-0.

[28] Robert Morris and Ken Thompson. "Password Security: A Case History". In: *COMMUNICATIONS OF THE ACM* 22 (1979), pp. 594–597.

[29] Adriana Esmeraldo de Oliveira, Gustavo Henrique Matos Bezerra Motta and Leonardo Vidal Batista. "A multibiometric access control architecture for continuous authentication". In: *Intelligence and Security Informatics (ISI), 2010 IEEE International Conference on*. 2010, p. 171. DOI: 10.1109/ISI.2010.5484746.

[30] Colin Percival. "Stronger Key Derivation Via Sequential Memory-Hard Functions". In: *BSDCan 2009: The Technical BSD Conference*. 2009.

[31] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321 (Informational). Updated by RFC 6151. Internet Engineering Task Force, Apr. 1992. URL: http://www.ietf.org/rfc/rfc1321.txt.

[32] Ron Rivest. *[Python-Dev] hashlib - faster md5/sha, adds sha256/512 support*. Ron Rivest officially confirms MD5 as broken. Dec. 2005. URL: http://mail.python.org/pipermail/python-dev/2005-December/058850.html (visited on 14/02/2012).

[33] Yu Sasaki and Kazumaro Aoki. "Finding Preimages in Full MD5 Faster Than Exhaustive Search". In: *EUROCRYPT*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 134–152. ISBN: 978-3-642-01000-2.

[34] IEEE Computer Society. *IEEE Standard for Local and metropolitan area networks – Part 21: Media Independent Handover Services*. Tech. rep. 2009.

[35] Frank Stajano. "Pico: No More Passwords!" In: *Security Protocols Workshop*. Ed. by Bruce Christianson et al. Vol. 7114. Lecture Notes in Computer Science. Springer, 2011, pp. 49–81. ISBN: 978-3-642-25866-4.

[36] Marc Stevens et al. "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate". In: *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '09. Santa Barbara, CA: Springer-Verlag, 2009, pp. 55–69. ISBN: 978-3-642-03355-1. DOI: 10.1007/978-3-642-03356-8_4. URL: http://dx.doi.org/10.1007/978-3-642-03356-8_4.

[37] Mark Stiegler. "An Introduction to Petname Systems". In: *Advances in Financial Cryptography* 2 (2005).

[38] Petter Taugbøl. *Lucidman.org*. Jan. 2012. URL: http://www.lucidman.org/ (visited on 01/09/2012).

[39] TazTag. *Mobility Products*. URL: http://taztag.com/index.php?option=com_content&view=article&id=104 (visited on 11/11/2012).

[40] Meltem Sönmez Turan et al. *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications*. Tech. rep. Gaithersburg, MD, United States, 2010.

[41] Kent Varmedal. "Cognitive entity authentication with Petname systems". MA thesis. Norway: University of Oslo, 2013.

[42] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions". In: *EUROCRYPT*. Ed. by Ronald Cramer. Vol. 3494. Lecture Notes in Computer Science. Springer, 2005, pp. 19–35. ISBN: 3-540-25910-4.

[43] Oxford Dictionary World of words. *What is the frequency of the letters of the alphabet in English?* URL: http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english (visited on 15/02/2012).

[44] Tao Xie and Dengguo Feng. *How To Find Weak Input Differences For MD5 Collision Attacks*. Cryptology ePrint Archive, Report 2009/223. http://eprint.iacr.org/. 2009.

Glossary

ACL The ACL is a list of authorizations. It describes the entities' access rights to protected objects. ACLs can be centric to users (i.e. an entry per user, listing his or her access rights to objects), realms (i.e. for each realm protected by the system, listing every entity that is to be approved access) or objects (i.e. for each object, specify what entities have access). 28.

Android Android is an operating system for mobile devices. It is based on Linux and is the most popular smartphone platform.

Authentication Assurance Level The level of assurance the system can have in the veracity of an identity when authenticated.

digest (also **hash, hash code)** The output of a hash function. 17.

hash *See* digest.

hash code *See* digest.

hash function A one-way function with specific properties. 17.

J2ME Java 2 Mobile Edition is a Java platform for mobile devices or embedded systems.

MD5 The MD5 Message-Digest Algorithm. 30.

MITM Describes the notion of an adversary somehow interfering, either actively (modifying content) or passively (snooping), on a communications channel. 25.

nonce A **n**umber used only **once**. 25.

realm Also *protection realm*. An environment in a computer network or file system (e.g. a directory). 22, 33.

user agent An agent (i.e. software) that acts on behalf of the user. Common user agents include web browsers, email clients and the like. 33.

Acronyms

AAL Authentication Assurance Level.

ACL Access Control List.

API Application Programming Interface.

HTTPd HTTP daemon (Apache HTTP Server).

J2ME Java 2 Mobile Edition.

KDF Key derivation function.

LUCIDMAN The Local User-Centric Identity Management project.

MAC Mandatory Access Control.

MITM Man In The Middle (attack).

MP-Auth Mobile Password Authentication [26].

OffPAD (*mostly*) Offline Personal Authentication Device.

OTP One-Time Pad, One-Time Password.

PAD Personal Authentication Device.

PIN Personal Identification Number.

qop Quality of Protection.

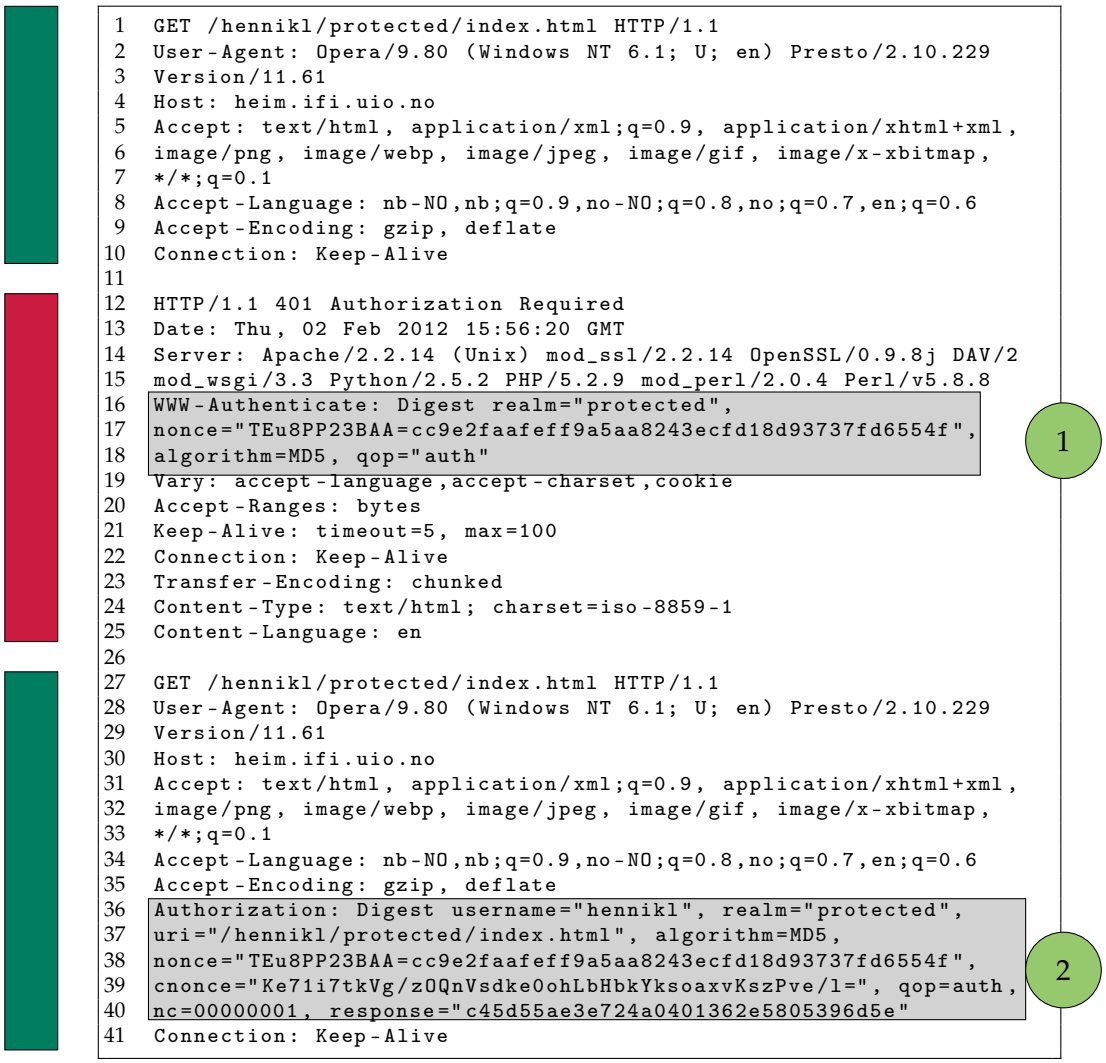
SSO Single Sign-On.

TOCTOU Time Of Check to Time Of Use.

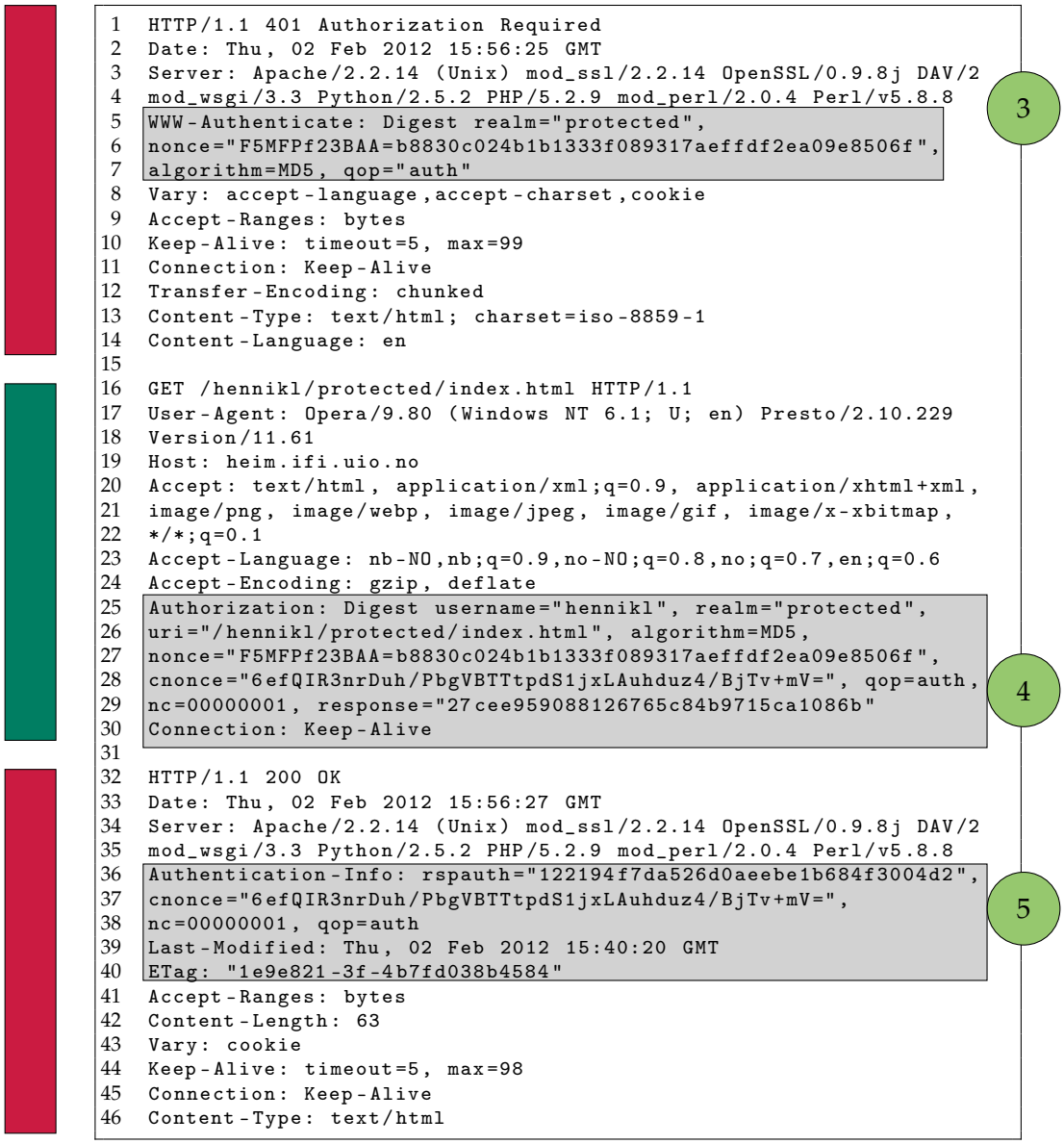
TOFU Trust On First Use.

Appendix A

Sample HTTP authentication



```
1  GET /hennikl/protected/index.html HTTP/1.1
2  User-Agent: Opera/9.80 (Windows NT 6.1; U; en) Presto/2.10.229
3  Version/11.61
4  Host: heim.ifi.uio.no
5  Accept: text/html, application/xml;q=0.9, application/xhtml+xml,
6  image/png, image/webp, image/jpeg, image/gif, image/x-xbitmap,
7  */*;q=0.1
8  Accept-Language: nb-NO,nb;q=0.9,no-NO;q=0.8,no;q=0.7,en;q=0.6
9  Accept-Encoding: gzip, deflate
10 Connection: Keep-Alive
11
12 HTTP/1.1 401 Authorization Required
13 Date: Thu, 02 Feb 2012 15:56:20 GMT
14 Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8j DAV/2
15 mod_wsgi/3.3 Python/2.5.2 PHP/5.2.9 mod_perl/2.0.4 Perl/v5.8.8
16 WWW-Authenticate: Digest realm="protected",
17 nonce="TEu8PP23BAA=cc9e2faafeff9a5aa8243ecfd18d93737fd6554f",
18 algorithm=MD5, qop="auth"
19 Vary: accept-language,accept-charset,cookie
20 Accept-Ranges: bytes
21 Keep-Alive: timeout=5, max=100
22 Connection: Keep-Alive
23 Transfer-Encoding: chunked
24 Content-Type: text/html; charset=iso-8859-1
25 Content-Language: en
26
27 GET /hennikl/protected/index.html HTTP/1.1
28 User-Agent: Opera/9.80 (Windows NT 6.1; U; en) Presto/2.10.229
29 Version/11.61
30 Host: heim.ifi.uio.no
31 Accept: text/html, application/xml;q=0.9, application/xhtml+xml,
32 image/png, image/webp, image/jpeg, image/gif, image/x-xbitmap,
33 */*;q=0.1
34 Accept-Language: nb-NO,nb;q=0.9,no-NO;q=0.8,no;q=0.7,en;q=0.6
35 Accept-Encoding: gzip, deflate
36 Authorization: Digest username="hennikl", realm="protected",
37 uri="/hennikl/protected/index.html", algorithm=MD5,
38 nonce="TEu8PP23BAA=cc9e2faafeff9a5aa8243ecfd18d93737fd6554f",
39 cnonce="Ke71i7tkVg/zOQnVsdke0ohLbHbkYksoaxvKszPve/l=", qop=auth,
40 nc=00000001, response="c45d55ae3e724a0401362e5805396d5e"
41 Connection: Keep-Alive
```


1 HTTP/1.1 401 Authorization Required
2 Date: Thu, 02 Feb 2012 15:56:25 GMT
3 Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8j DAV/2
4 mod_wsgi/3.3 Python/2.5.2 PHP/5.2.9 mod_perl/2.0.4 Perl/v5.8.8
5 WWW-Authenticate: Digest realm="protected",
6 nonce="F5MFPf23BAA=b8830c024b1b1333f089317aeffdf2ea09e8506f",
7 algorithm=MD5, qop="auth"
8 Vary: accept-language,accept-charset,cookie
9 Accept-Ranges: bytes
10 Keep-Alive: timeout=5, max=99
11 Connection: Keep-Alive
12 Transfer-Encoding: chunked
13 Content-Type: text/html; charset=iso-8859-1
14 Content-Language: en
15
16 GET /hennikl/protected/index.html HTTP/1.1
17 User-Agent: Opera/9.80 (Windows NT 6.1; U; en) Presto/2.10.229
18 Version/11.61
19 Host: heim.ifi.uio.no
20 Accept: text/html, application/xml;q=0.9, application/xhtml+xml,
21 image/png, image/webp, image/jpeg, image/gif, image/x-xbitmap,
22 */*;q=0.1
23 Accept-Language: nb-NO,nb;q=0.9,no-NO;q=0.8,no;q=0.7,en;q=0.6
24 Accept-Encoding: gzip, deflate
25 Authorization: Digest username="hennikl", realm="protected",
26 uri="/hennikl/protected/index.html", algorithm=MD5,
27 nonce="F5MFPf23BAA=b8830c024b1b1333f089317aeffdf2ea09e8506f",
28 cnonce="6efQIR3nrDuh/PbgVBTTtpdS1jxLAuhduz4/BjTv+mV=", qop=auth,
29 nc=00000001, response="27cee959088126765c84b9715ca1086b"
30 Connection: Keep-Alive
31
32 HTTP/1.1 200 OK
33 Date: Thu, 02 Feb 2012 15:56:27 GMT
34 Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8j DAV/2
35 mod_wsgi/3.3 Python/2.5.2 PHP/5.2.9 mod_perl/2.0.4 Perl/v5.8.8
36 Authentication-Info: rspauth="122194f7da526d0aeebe1b684f3004d2",
37 cnonce="6efQIR3nrDuh/PbgVBTTtpdS1jxLAuhduz4/BjTv+mV=",
38 nc=00000001, qop=auth
39 Last-Modified: Thu, 02 Feb 2012 15:40:20 GMT
40 ETag: "1e9e821-3f-4b7fd038b4584"
41 Accept-Ranges: bytes
42 Content-Length: 63
43 Vary: cookie
44 Keep-Alive: timeout=5, max=98
45 Connection: Keep-Alive
46 Content-Type: text/html
3
4
5

1. The server (●) requests the user (●) to authenticate, as it is accessing an *authorized only* realm. The `WWW-Authenticate` header field (highlighted) contains information for the user to decide what credentials apply at the location (realm) and a value for avoiding replay attacks etc. (nonce).

2. The user has now entered his credentials and by using the information from `WWW-Authenticate`, the user agent/browser has created the response value that is passed back.

3. Here the server again requests the user to authenticate; this time because the credentials entered were erroneous. A new nonce is generated for the user to try again.

4. The user now supplies a new set of credentials and a new response is created.

5. The `200 OK` from the server means that authorization is completed successfully. The resource the user requested follows after the header (not shown). Also, the Authentication-Info field contains information about the past authentication.